Trabajo Fin de Grado
Grado en Ingeniería Informática
Ingeniería de Computadores

Implementación de prebuscadores de cache para el procesador RISC-V DRAC

Implementation of cache prefetchers for the RISC-V DRAC processor

Autor:

Javier Salamero Sanz

Directores:

Pablo Ibañez Marín

Jesús Javier Resano Ezcaray

Escuela de Ingeniería y Arquitectura (EINA)
Universidad de Zaragoza

2021

Agradecimientos

En esta sección quiero agradecer a mis directores el esfuerzo y dedicación puestos en el seguimiento y dirección de este trabajo de fin de grado. También quiero agradecer a Agustín Navarro pues, a pesar de no ser tutor de mi TFG, me ha ayudado en todo momento en la fase de simulación, proporcionándome material o cuando han surgido problemas en el simulador. Además, agradecer Guillem López y Víctor Soria del CNS por ayudarme a trabajar con el procesador Sargantana. Por último, dar las gracias a mi familia por la paciencia y comprensión demostrada.

Resumen

El tiempo de acceso a memoria es un elemento clave en el rendimiento de todo procesador moderno. Un mecanismo que permite reducir este tiempo es la prebúsqueda, que consiste en mover datos a las memorias cache antes de que sean solicitados por el procesador. Existen diversas aproximaciones de este mecanismo, en este trabajo se han explorado algunas de ellas.

Por otra parte, la Unión Europea ha marcado como objetivo estratégico diseñar y fabricar procesadores de propósito general a través de la iniciativa EPI. La motivación tras esta iniciativa es la necesidad de desarrollar procesadores de diseño propio capaces de satisfacer una creciente demanda en computación a gran escala, a la vez que disminuir la dependencia de mercados externos. En el contexto de EPI, surgen muchos proyectos derivados, como el proyecto DRAC, cuyo objetivo es el desarrollo de un procesador de propósito general basado en la tecnología RISC-V y sus aceleradores, liderado por el CNS.

Este trabajo fin de grado consiste en el estudio, simulación, implementación e integración de un prebuscador de instrucciones en un procesador real, con el deseo de que forme parte del chip que finalmente se fabrique. El objetivo principal en este TFG es adquirir un visión global de todas las fases involucradas en el desarrollo de un módulo en un procesador. El procesador elegido es Sargantana, una de las líneas de trabajo dentro del proyecto DRAC. Se han estudiado diversos mecanismos de prebúsqueda, de los cuales se han analizado sus prestaciones a través de la simulación, utilizando *benchmarks* SPEC como aplicaciones de referencia. Finalmente, se ha escogido un prebuscador para su implementación en RTL utilizando el lenguaje de descripción VHDL. Además de verificar el correcto funcionamiento de la implementación, se ha comprobado que sea rápida y de bajo coste en silicio. Para ello, se han analizado los resultados de síntesis en una FPGA. Por último, se ha trabajado en la integración con Sargantana.

Durante el desarrollo de este trabajo se han puesto en práctica todos los conocimientos en el área de arquitectura adquiridos durante el grado. Tras él, se ha demostrado la aceleración que supone la prebúsqueda en un procesador y se ha comprendido la metodología a seguir en proyectos de la misma índole, además de descubrir su complejidad subyacente. Como finalización, se han planteado diversas vías para la continuación de este trabajo en un futuro. Se puede encontrar en `https://github.com/Haz99/Sargantana-TFG` el repositorio del proyecto con los fuentes, *scripts* y casos de estudio utilizados.

Índice

1. Introducción

El objetivo principal de este trabajo de fin de grado (TFG) es abordar todas las fases en el desarrollo de un módulo lógico, cuya finalidad es ser integrado en un microprocesador moderno. Este módulo, concretamente, se trata de un mecanismo de prebúsqueda en memorias cache, llamado prebuscador. Los prebuscadores tienen el objetivo de reducir el número de ciclos necesarios desde que el procesador realiza un acceso a memoria, hasta que obtiene el elemento solicitado (latencia). Como se verá más adelante, puede tratarse de un dato o una instrucción, y la forma de acelerar este proceso es solicitando los elementos antes de necesitarlos. Esta reducción de latencia permite un incremento directo en el rendimiento del procesador, donde la utilización de memoria viene siendo, ya desde hace décadas, un cuello de botella destacado [1], [2].

El contexto en el que se entiende este TFG parte de la iniciativa *European Processor Initiative* (EPI) [3]. EPI es un proyecto de la Unión Europea (UE) surgido de la necesidad de disponer de procesadores de altas prestaciones de fabricación y diseño locales en un futuro cada vez más próximo. El objetivo de la UE es desarrollar procesadores de propósito general y bajo consumo para computación a gran escala. Este proyecto pretende generar un fuerte modelo económico conjunto entre sus miembros que permita reducir la dependencia actual que se tiene de mercados extranjeros, a la vez que beneficiarse de las oportunidades que generaría este tipo de industria, como por ejemplo satisfacer las necesidades de los fabricantes europeos de automóviles [3]. Uno de estos proyectos relacionados, es el proyecto DRAC (*Designing RISC-V-based Accelerators for next generation Computers*), liderado por el el Centro Nacional de Supercomputación (CNS) [4], en colaboración con otras universidades. El objetivo de DRAC es el diseño, la implementación y la fabricación de un procesador de propósito general basado en la arquitectura RISC-V y sus aceleradores [5]. De este proyecto surge el procesador Sargantana, en el cual se pretende la integración del módulo de prebúsqueda que se va a desarrollar en este trabajo.

Las fases que intervienen en el desarrollo de este módulo son comunes a cualquier otro componente de un procesador. Primero, se ha realizado un estudio teórico del mecanismo que se pretende integrar, en este caso, se han valorado distintos diseños de prebuscadores que se han ido publicando a lo largo de los años, tanto de datos como de instrucciones. De estos diseños, se ha analizado su comportamiento y valorado sus características, ventajas e inconvenientes.

La siguiente fase es la simulación, la cual consiste en desarrollar un simulador temporal de un procesador lo más parecido posible a Sargantana. En él, se ejecutarán diversos *benchmarks* (ver Anexo A) de los que se extraerán métricas de rendimiento, con diferentes prebuscadores que permitan analizar su eficacia y eficiencia teniendo en cuenta el coste de implementación. En este trabajo se pretende utilizar los *benchmarks* SPEC CPU 2017 [6], los cuales son globalmente aceptados para la comparación de procesadores.

La última fase es la implementación e integración, donde se diseñará y verificará en RTL (ver Anexo A) el módulo de prebúsqueda de forma independiente, utilizando la herramienta Modelsim, utilizada en asignaturas del grado para la misma finalidad. Se usará el lenguaje de especificación VHDL, el cual permite diseñar circuitos digitales de forma textual [7]. Una vez comprobado el correcto funcionamiento del módulo, se intentará acoplar al procesador real, también especificado en un lenguaje similar, Verilog.

A continuación, se describirá brevemente la organización de la memoria. Tras esta introducción, en la Sección 2 se abordará el contexto del proyecto, donde se incluye de forma más detallada la motivación tras este trabajo, así como una breve explicación del funcionamiento del procesador

Sargantana, en el que se va a integrar la prebúsqueda, y explicaciones detalladas sobre las características de los prebuscadores que se van a estudiar. La Sección 3 trata sobre la fase de simulación, en la que se explica la metodología seguida, las herramientas utilizadas, además de las aplicaciones y casos de estudio analizados. Esta sección concluye con los resultados conseguidos y una reflexión sobre los mismos. En la Sección 4 se detalla los pasos seguidos en la fase de integración e implementación, consideraciones sobre el diseño y análisis del coste de la propuesta. La memoria finaliza con las conclusiones extraídas de este trabajo y se exploran posibles vías de continuación.

Por último, se recomienda encarecidamente la lectura de los anexos, pues en ellos se incluye un glosario de términos, ejemplos completos del funcionamiento de cada prebuscador, la función de cada *benchmark* utilizado, todos los resultados obtenidos, detalles ortogonales en la prebúsqueda de datos, etc. No obstante, se indicará en la memoria cuando sea recomendable su consulta.

2. Contexto

Desde la Unión Europea se ha estudiado la importancia estratégica de disponer de tecnologías propias en procesadores de altas prestaciones (HPC). Los motivos residen en el deseo de poseer máquinas con potencia de cómputo exaescala (ver glosario en el Anexo A) de 2022 en adelante, las necesidades de potencia de cómputo en conducción autónoma y la importancia de los procesadores de bajo consumo en servidores y sistemas Cloud [3]. Es por ello que se han establecido una serie de objetivos, algunos de ellos son: conseguir la independencia tecnológica en procesadores HPC de mercados asiáticos y americanos, fabricar una máquina con potencia exaescala basada en procesadores de diseño europeo para 2023, establecer un modelo económico sólido a largo plazo mas allá de HPC y satisfacer las necesidades computacionales en la industria automovilística europea [3], [8].

Bajo estas necesidades surge la iniciativa "European Processor Initiative (EPI)"[3], cuya primera fase EPI SGA1(Specific Grant Agreement 1) [8] marca como objetivo "diseñar y desarrollar los primeros procesadores europeos de sistema en chip y de aceleración para la computación de altas prestaciones. Ambos elementos se aplicarán y validarán en un prototipo que servirá de base para una máquina completa a exaescala"[8]. Esta iniciativa consiste en el desarrollo de un procesador de propósito general de bajo consumo con un enfoque hacia la paralelización masiva, centrándose en aspectos como la eficiencia energética, la seguridad y el uso de aceleradores específicos. Con ello se pretende conseguir una plataforma común que agrupe distintas tecnologías hacía un mismo objetivo, teniendo como base compartida la arquitectura, la metodología de diseño y el enfoque energético[3], [8].

Este proyecto cuenta con 28 socios de 10 países distintos dentro de la Unión Europea, entre los que se encuentra el Centro Nacional de Supercomputación [3], [4], de ahora en adelante, CNS. Siguiendo este objetivo, en el CNS existen varios proyectos, entre los que destaca para este trabajo de fin de grado el proyecto DRAC ("*Designing RISC-V-based Accelerators for next generation Computers*"), donde se "diseñará, verificará, implementará y fabricará un procesador de propósito general de alto rendimiento que incorporará diferentes aceleradores basados en la tecnología RISC-V [9], [10], con aplicaciones específicas en el campo de la seguridad, la genómica y la navegación autónoma"[5]. Dentro de DRAC, existen varias líneas de trabajo, entre las cuales se desarrolla el procesador Sargantana, sobre el cual se ha trabajado en este proyecto.

2.1. Sargantana

Sargantana es un procesador de la arquitectura RISC-V actualmente en desarrollo y perteneciente al proyecto DRAC[5], sobre el que se va a trabajar en este trabajo fin de grado. El objetivo principal que se está desarrollando desde el CNS es integrar en la etapa de ejecución de este procesador de propósito general un acelerador vectorial RISC-V diseñado para el procesador Avispado de *Semidynamics*[11]. Este tipo de aceleradores son una de las líneas de investigación del proyecto EPI.

Se trata de un procesador segmentando en cinco etapas (búsqueda, decodificación, lectura de registros, ejecución y consolidación), con lanzamiento de instrucciones en orden y finalización en fuera de orden. Cuenta con cinco caminos de ejecución (salto, acceso a memoria, aritmético, multiplicación y división), además, se pretende integrar el acelerador vectorial ya mencionado. La jerarquía de memorias está compuesta por una cache L1 dedicada para instrucciones (L1i) de 16 kB (64 conjuntos y 4 vías), otra dedicada para datos (L1d) del mismo tamaño, y una única cache L2 de 64 kB (128 conjuntos y 8 vías) compartida y que debe ser coherente con las L1, todas ellas con política de reemplazo aleatoria. La mayoría de los componentes de memoria han sido usados de *LowRisc System on chip (SoC) version 0.2 Untethered*[12].

A continuación, se va a realizar un breve repaso de la funcionalidad de cada etapa, las cuales están gobernadas por una unidad de control (UC):

Búsqueda: En esta etapa se realiza la petición a la memoria cache de instrucciones, con la dirección indicada por el contador de programa (PC). Con esta petición, la cache devuelve la instrucción a ejecutar, que se propaga a la siguiente etapa. Además, como los saltos se resuelven más adelante, en lugar de tener vacías el resto de etapas del segmentado hasta su resolución, se especula con la ejecución que va a tomar el programa. El modulo encargado de predecir la siguiente instrucción es el predictor de saltos, el cual decide si la instrucción se trata de un salto y qué decisión tomar al respecto, tomar salto o no tomar. Sargantana utiliza un predictor local de saltos bimodal (*2-bit predictor*), el cual cuenta con dos bits para codificar el estado del salto: tomar débil, tomar fuerte, no tomar débil y no tomar fuerte. El estado se va ajustando conforme se resuelven los saltos [13]. En Sargantana, esta etapa se ha dividido en dos; búsqueda 1 y búsqueda 2, no obstante, conceptualmente se trata de una etapa de búsqueda con la funcionalidad descrita anteriormente pero segmentada en dos ciclos. El motivo de esta división es que la cache de instrucciones no es capaz de responder a una petición en un ciclo. En la Figura 2.1 aparece como una única etapa pero debe considerarse su segmentación.

Decodificación: Consiste en averiguar qué instrucción se quiere ejecutar. En esta etapa, la UC genera las señales necesarias a propagar al resto de etapas para conseguir la ejecución de la instrucción.

Lectura de registros: A continuación, la instrucción atraviesa la etapa de lectura de registros donde, como su propio nombre indica, se lee el valor de los registros fuente utilizados por la instrucción. Se recuerda que la arquitectura es **RISC**, lo cual indica que únicamente se puede operar directamente sobre datos en registros y no directamente en memoria [13], [14]. Por otro lado, en esta etapa se propagan las instrucciones en orden y se decide si pueden ser lanzadas o no bajo dos criterios: la existencia de riesgos estructurales (camino de ejecución ocupados, estructuras llenas, coincidencia en el ciclo de consolidación con otras instrucciones, etc) o riesgos de datos (operandos fuentes no preparados). Esta decisión la toma una estructura llamada *scoreboard*[13], la cual contiene la lógica necesaria para ello. Por último, en esta etapa se produce el renombre de registros en las instrucciones[13], sustituyendo los

registros lógicos que ofrece la arquitectura RISC-V (32 de propósito general)[10] por registros físicos del procesador (64). Este mecanismo permite a varias instrucciones utilizar el mismo registro virtual destino simultáneamente.

Ejecución: En esta etapa las instrucciones atraviesan las unidades funcionales (UF) de sus caminos correspondientes.

Consolidación: La etapa de consolidación se encarga de escribir los resultados en el banco de registros, notificar la resolución del salto a la UC, tratar las excepciones, etc. Como las instrucciones, a pesar de lanzarse en orden, pueden terminar en desorden (no todas ocupan el mismo tiempo en etapa de ejecución), se utiliza una estructura que permite consolidarlas nuevamente en orden, llamada *re-order buffer* (ROB)[13]. En esta estructura, compuesta de una entrada por cada instrucción en vuelo, las instrucciones se insertan en orden en la etapa de decodificación. Los resultados se almacenan en el ROB conforme se producen en la fase de ejecución, para poder suministrarlos a instrucciones posteriores. Por último, las entradas más antiguas se consumen en etapa de consolidación y se escribe en el banco de registros consolidados. En este procesador, es posible consolidar hasta dos instrucciones por ciclo.

Esta es la caracterización general sobre el procesador en el que se pretende trabajar. La Figura 1 presenta un diagrama del camino de datos del procesador que ayuda a visualizar su funcionamiento.

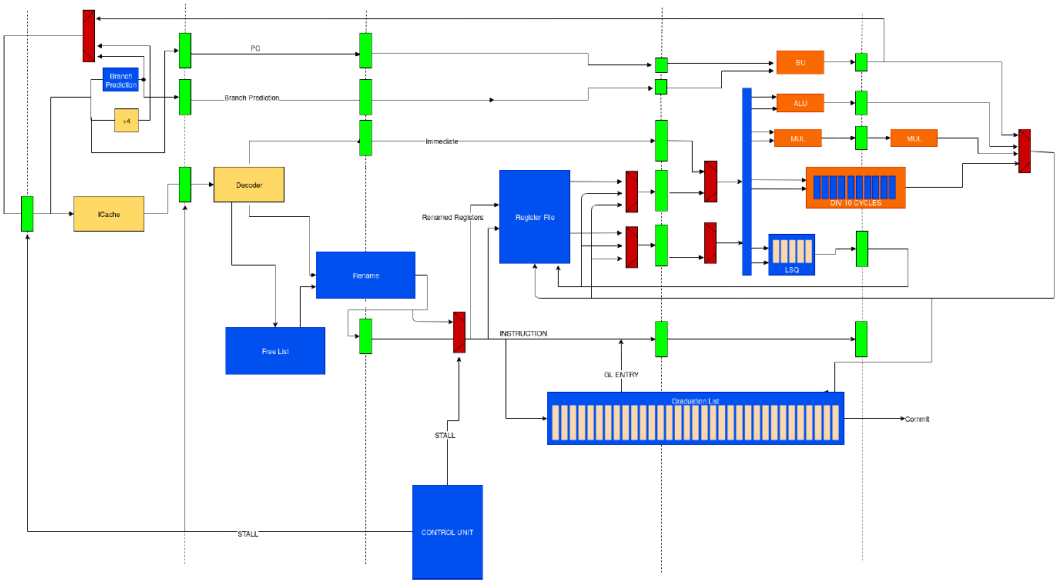

Figura 1: Diagrama del camino de datos de Sargantana proporcionado por el CNS[4]

2.2. Prebúsqueda

Los accesos a memoria son uno de los elementos más importante de todo procesador. Disponer de una latencia de acceso a instrucciones y datos lo más baja posible es fundamental para alcanzar rendimientos óptimos. Es por ello que los procesadores modernos incorporan una jerarquía de memorias más pequeñas llamadas caches, situadas entre el camino de datos (ver Anexo A) y la memoria principal. Este modelo permite reducir la latencia de acceso media y, por tanto, disminuir los tiempos de ejecución de forma sustancial [1], [15].

Una forma de acelerar todavía más este proceso consiste en mover los datos de memoria a las caches antes de necesitarlos. Este movimiento, generalmente se realiza agrupado bytes consecutivos de tamaño fijo en bloques (ver Anexo A). Este procedimiento se conoce como prebúsqueda. Podemos clasificar los prebuscadores existentes en varios ejes ortogonales entre sí [1], [2].

Por un lado, existe la clasificación en función de si la prebúsqueda es visible o no a la arquitectura del lenguaje máquina, ALMA (*instruction set architecture, ISA*, en inglés) (ver Anexo A). La prebúsqueda *software*, visible en la ISA, consiste en insertar instrucciones dedicadas a traer bloques de memoria con antelación. Por el contrario, la prebúsqueda *hardware* requiere soporte a nivel de micro-arquitectura del procesador. El programador puede estar utilizando prebúsqueda *hardware* sin ser consciente de ello, puesto que es transparente para él, aunque puede configurarse si sabe de su existencia [2].

Otra clasificación posible es en función de qué se trae con antelación: instrucciones o datos. Esta clasificación tiene implicaciones en la forma de predecir los bloques cache con antelación [1], [2].

Como aclaración, en este trabajo se van a analizar mecanismos de prebúsqueda hardware de datos y de instrucciones, los cuales trabajan exclusivamente a nivel de cache L1i o L1d, aunque las peticiones que generen acaben desencadenando peticiones en el resto de niveles de la jerarquía. Por último, alguno de ellos pertenece a diseños de los años 80 y 90 (Stride, secuencial) [1], [2], sin embargo, pese a los avances que se han realizado desde entonces en micro-arquitectura, estos modelos o derivados siguen siendo utilizados en procesadores actuales como el *Intel Xeon Gold 5120 (Skylake-SP)* [16]-[18] o la familia de procesadores *Power9* de IBM [19].

2.2.1. Prebuscadores de instrucciones

La prebúsqueda de instrucciones permite que la instrucción se encuentre en L1i antes de que el propio contador de programa (PC) la solicite. Aunque diversas aproximaciones para lograrlo, en este trabajo se han explorado dos de ellas: ***One block lookahead* (OBL)** y **prebúsqueda guiada por predictor de saltos**.

2.2.1.1. One block lookahead en instrucciones

Una de las soluciones más sencillas es la **prebúsqueda secuencial**, es decir, solicitar el siguiente elemento al que se está accediendo por petición de la etapa de búsqueda (petición por demanda). Este tipo de prebúsqueda en instrucciones aprovecha la localidad espacial (ver Anexo A) que se encuentra en el orden de ejecución de todo programa (recordar que las instrucciones se ejecutan secuencialmente salvo que se produzca un salto) [2].

"Los mecanismos más simples de prebúsqueda secuencial son variaciones sobre el concepto *one block lookahead* (OBL)"[2], el cual consiste en prebuscar el bloque cache siguiente al que se está solicitando. En el Anexo C.2 se adjunta un ejemplo de funcionamiento. Las desventajas principales de este tipo de implementación son, por un lado, que el número de peticiones a memoria es *2N*, siendo *N* el número de instrucciones que se ejecutan y, por otro, la poca agresividad de la prebúsqueda, pues sólo se prebusca un bloque. Por el contrario, la ventaja de ser poco agresivo es la limitación en la polución (ver Anexo A) de las caches [1], [2].

2.2.1.2. Prebuscador guiado por predictor de saltos

Una de las preguntas a las que debe responder todo mecanismo de prebúsqueda de instrucciones es cual es la próxima instrucción que se va a solicitar a memoria y ejecutar. Por fortuna, hay un módulo presente en todo procesador moderno cuya finalidad es exactamente predecir cual será la siguiente instrucción a ejecutar, el predictor de saltos [13], el cual tiene una tasa de acierto superior al 90 % en diseños contemporáneos. La idea de este prebuscador es preguntar al predictor de saltos la siguiente instrucción que se va a ejecutar para pedirla con antelación [20], [21].

Por otro lado, el diseño propuesto desacopla totalmente las instrucciones solicitadas por la etapa de búsqueda de la prebúsqueda. Las peticiones de direcciones generadas por prebúsqueda y por etapa de búsqueda coexisten paralelamente en caminos independientes. En la Figura 2 se muestra la estructura de este prebuscador donde puede verse esta división.

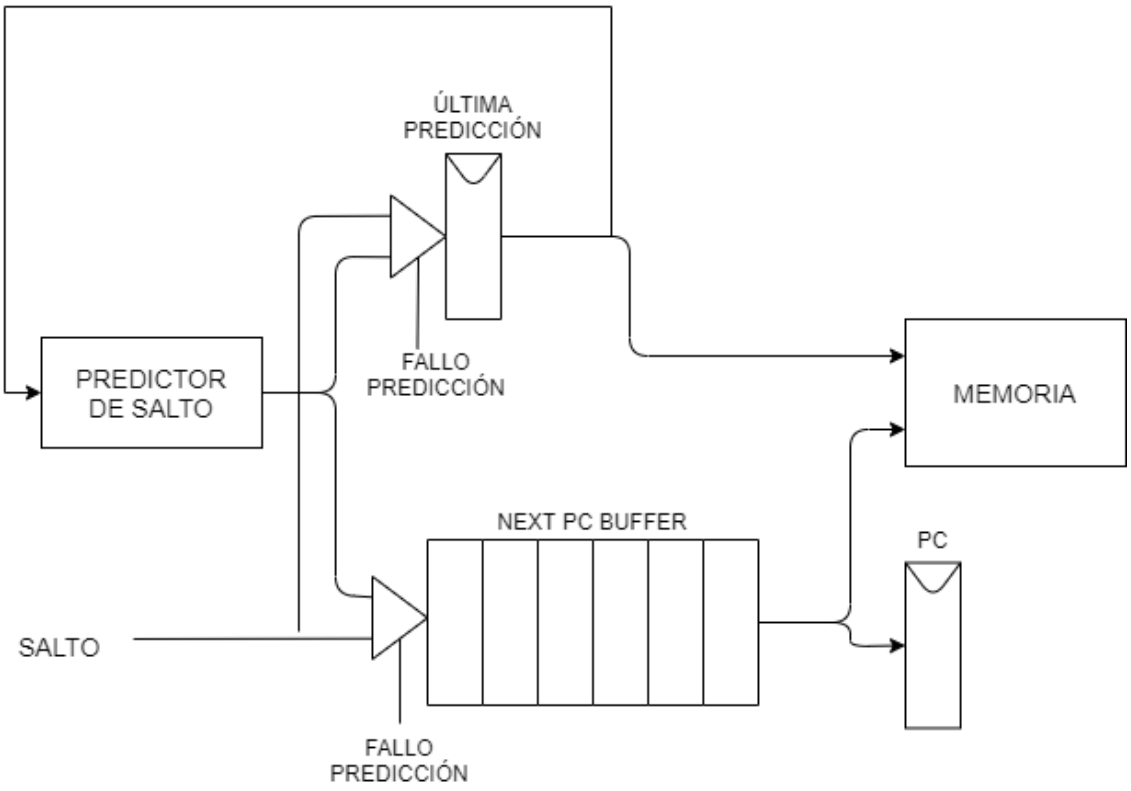

Figura 2: Estructura del prebuscador guiado por predictor de saltos

Con esta nueva estructura se generan peticiones simultáneas a memoria, desde el camino de prebúsqueda (donde se ubica el registro *última predicción*) y desde el camino correspondiente a la etapa de búsqueda (donde se ubica la estructura *next pc buffer*). El requisito en memoria para atender los dos tipos de peticiones simultáneamente es la existencia de 2 puertos. A continuación, se detalla el funcionamiento de este prebuscador.

Este tipo de prebuscador se integra en la etapa de búsqueda. Tiene dos caminos a memoria funcionando simultáneamente:

- El *predictor de salto* recibe la dirección de una instrucción y genera la dirección de la instrucción que presumiblemente se ejecutará tras ella. Cada ciclo, la dirección generada por el predictor se almacena en el registro *última predicción*, y este registro alimentará al predictor durante el siguiente ciclo. Estas direcciones forman la secuencia de prebúsqueda. Cada ciclo, una nueva dirección de prebúsqueda es suministrada a la cache de instrucciones, generando una petición.

- Por otro lado, en el camino inferior de la Figura 2, la etapa de búsqueda realiza peticiones a *memoria* con la secuencia de direcciones de las instrucciones que se van a ejecutar. En un procesador sin prebúsqueda, esta secuencia es generada por el predictor de saltos, por lo tanto en el procesador utilizado coincide con la secuencia de prebúsqueda. Este diseño usa una estructura FIFO para almacenar estas direcciones (*next pc buffer*). Cuando se produce un fallo de cache en la etapa de búsqueda, se detiene el lanzamiento de nuevas peticiones desde este camino hasta que *memoria* es capaz de proporcionar la instrucción. Mientras se resuelve un fallo, en *next pc buffer* se van acumulando las direcciones de las próximas instrucciones a ejecutar producidas por el *predictor de salto*.

Existen dos excepciones en este comportamiento:

- Si se tardan muchos ciclos en resolver el fallo, es posible que *next pc buffer* se llene de direcciones, en ese caso se detiene su carga y la del registro *última predicción*, para no perder instrucciones en la secuencia de ejecución del programa.

- En caso de un fallo de predicción, se eliminan las direcciones almacenadas en *next pc buffer* (puesto que corresponden a un camino de ejecución incorrecto) y se inserta en dicha estructura y *última predicción* la dirección *salto*. Tras este ciclo en el que no se generan peticiones a *memoria*, se retoma el funcionamiento habitual.

Este prebuscador presenta una mayor agresividad en la prebúsqueda respecto a OBL. Además, se presupone una mayor precisión, y por tanto menor polución, debido a la alta tasa de acierto del predictor de saltos.

Por último, debido al menor volumen de peticiones en la memoria cache de instrucciones, se asume un diseño con un único puerto a memoria. En este caso, se da prioridad a las peticiones procedentes del camino de búsqueda. Se proporciona un ejemplo completo para este caso en el Anexo C.3

2.2.2. Prebuscadores de datos

Las direcciones de los datos en memoria a los que acceden las instrucciones presentan patrones mucho más variados que los que general el contador de programa (PC). Los accesos son secuenciales salvo cuando se producen saltos. Por ello, existen una gran variedad de mecanismos de prebúsqueda de datos [2]. En este trabajo, se han analizado tres diseños alternativos.

Además del diseño del propio prebuscador, intervienen otras características ortogonales como la política en el desencadenante de prebúsqueda, por ello se recomienda la lectura del Anexo E.

2.2.2.1. One block lookahead en datos

Al igual que ocurre con la prebúsqueda de instrucciones, la aproximación más simple consiste en solicitar el bloque siguiente al demandado. Suele funcionar relativamente bien, debido a que aprovecha la localidad espacial de los datos [2], y es sencillo de implementar. Existen subvariantes, como solicitar el bloque anterior si se solicita un dato de la mitad inferior de bloque, para permitir prebúsqueda de recorridos descendentes. Presenta las mismas ventajas e inconvenientes que los ya mencionados en el OBL de instrucciones.

2.2.2.2. Prebuscador stride

Otra solución se basa en intentar encontrar patrones de acceso repetitivos a datos en memoria para cada instrucción. Por ejemplo, en el recorrido de un vector, la misma instrucción accede a las distintas direcciones que componen los elementos del vector. La distancia entre la dirección de acceso de un elemento y el siguiente siempre es fija, en este caso el tamaño del elemento del vector. Asimismo ocurre con otro tipo de estructuras como el recorrido de matrices. Llamamos *stride* a esta distancia. Estos patrones de acceso con *stride* constante son bastante frecuentes, y ser capaz de aprenderlos y predecirlos permite ampliar la cobertura (ver Anexo A) de la prebúsqueda [2].

Es la búsqueda y aprendizaje de estos patrones fijos el objetivo del prebuscador stride. Como se muestra en la Figura 3, este prebuscador se compone de cuatro señales de entrada y dos de salida.

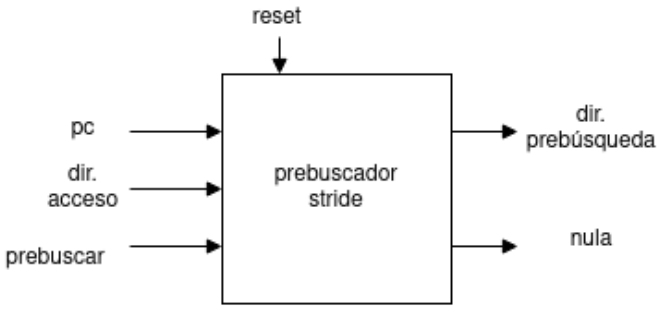

Figura 3: Bloque del prebuscador de datos stride

Señales de entrada:

- Reset: Señal de vaciado de estructuras internas.

- Pc: El contador de programa es necesario para identificar la instrucción que hace el acceso.
- Dirección acceso: Dirección del dato solicitado por la instrucción.
- Prebuscar: Señal de control que le indica al prebuscador que el resto de entradas son válidas.

Señales de salida:

- Dirección prebúsqueda: Dirección del dato a prebuscar.
- Nula: Señal de control que le indica al procesador si la dirección de prebúsqueda es válida.

La estructura interna se muestra en la Figura 4. El funcionamiento del prebuscador se organiza entorno a la *Tabla Stride* o *reference prediction table (RPT)* [2], la cual es indexada con N bits de PC. Cada entrada de RPT corresponde al seguimiento de una instrucción de acceso a memoria.

Campos de una entrada:

- Tag: M bits del pc de la instrucción almacenada.
- Last: La dirección del último dato accedido por la instrucción de la entrada.
- Stride: Distancia en bytes entre las direcciones (accesos) del patrón.
- Trust: Codificación de la confianza que se tiene en el patrón encontrado. Esta codificación es un valor numérico correspondiente a los estados: *patrón no encontrado* (0), *patrón encontrado con poca confianza* (1) y *patrón confiable* (2). Esta confianza indica cuando hay que lanzar una prebúsqueda, y la distinción entre los dos estados con patrón encontrado sirven para distinguir el comportamiento de la prebúsqueda en caso de fallo en el patrón.
- Válido: Indica que el contenido de la entrada tiene validez.

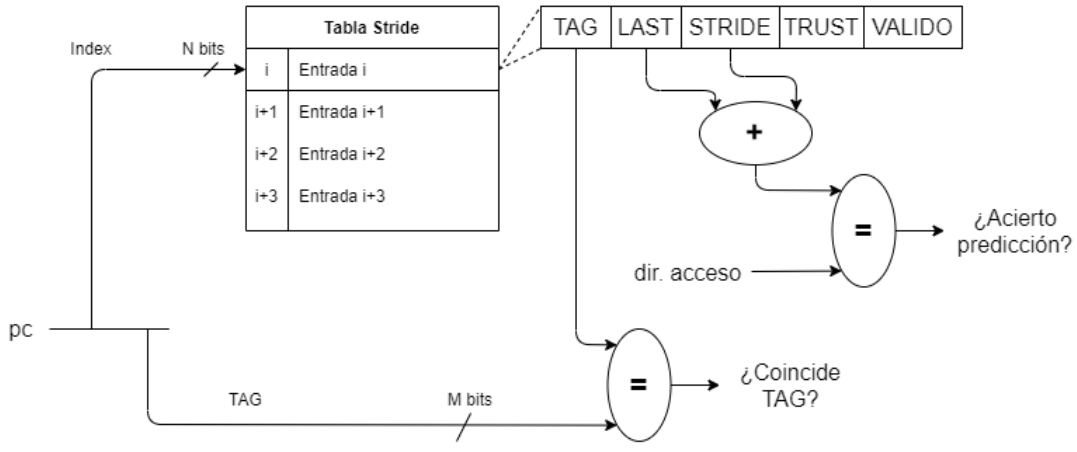

Figura 4: Estructura interna del prebuscador de datos stride

Por supuesto, varias instrucciones pueden coincidir en los N bits usados para seleccionar una entrada de RPT, es por ello que el campo *tag* almacena otros M bits para diferenciarlas. No es necesario que M+N sea igual a la longitud de PC, N está delimitado por el tamaño de la tabla, y con una longitud razonable de M se puede reducir el coste de almacenamiento y las coincidencias

son mínimas.

En la sección C.4 se proporciona un ejemplo práctico del funcionamiento del prebuscador, el cual se va a explicar a continuación. Si la entrada es válida y *tag* coincide con el almacenado, se predice la dirección de acceso de la instrucción, es decir, *last* + *stride*. Si es igual a la la señal *dirección de acceso* de entrada quiere decir que se ha acertado el patrón, se aumenta la confianza y se prebusca el siguiente elemento del patrón, *dirección de acceso* + *stride*. Además, por cada acceso se actualiza el campo *last*. Si por el contrario, la predicción falla, se reduce la confianza. Si el estado era *patrón encontrado con poca confianza* o *patrón no encontrado*, no se prebusca nada y se intenta encontrar el nuevo patrón. Por el contrario, si el estado era *patrón confiable*, se prebusca el siguiente elemento al accedido según el viejo patrón a partir de la dirección de acceso, pero se reduce la confianza a *patrón encontrado con poca confianza*, de forma que si se vuelve a fallar se desechará el valor *stride*. De esta forma se consiguen prebuscar patrones fijos que presenten algún salto irregular, pero mantengan el *stride* en la mayoría de casos.

Las ventajas de este prebuscador respecto al secuencial es la generalidad. En el caso del secuencial solo se trae el bloque contiguo al demandado, mientras que el prebuscador stride es capaz de prebuscar patrones de acceso con más de un bloque de distancia entre elementos. Por contra, es más costoso de implementar, puesto que se necesita gestionar la lógica de las estructuras internas, no permite detectar patrones de acceso de tamaño variable, y no es muy agresivo puesto que solo se prebusca el siguiente elemento del patrón.

2.2.2.3. Prebuscador adaptativo

Una evolución del prebuscador anterior es el prebuscador adaptativo, el cual es una simplificación del prebuscador multinivel propuesto por [22], diseñado por investigadores del *Departamento de Informática e Ingeniería de Sistemas*. Es una simplificación en el sentido de que solo prebusca en el nivel L1d de cache y no detecta patrones de acceso variables sino fijos como el Stride. Se busca ser más agresivo, para ello se pedirá más de un elemento en el patrón encontrado si la confianza es alta, característica conocida como grado de prebúsqueda (ver Anexo A). Además, se tiene control de los elementos ya prebuscados para no realizar peticiones duplicadas.

Puesto que es un diseño derivado del prebuscador Stride, solo se van a explicar aquellos elementos nuevos o que hayan variado en el modelo. En la Figura 5, se observa que las señales de entrada son las mismas, pero se añaden dos nuevas señales de salida.

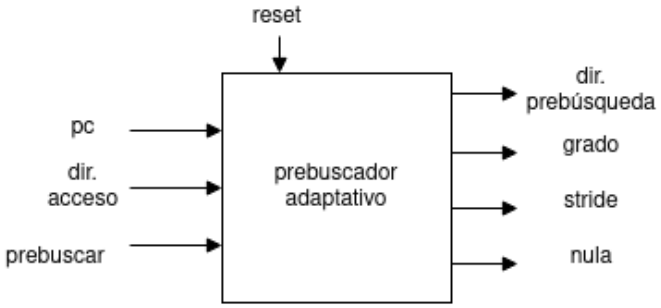

Figura 5: Bloque del prebuscador de datos adaptativo

Nuevas señales:

- Grado: Como se puede prebuscar más de un elemento, es necesario indicar cuantos son.
- Stride: Por la misma razón, se indica la distancia en bytes entre los elementos. En este caso dir. prebúsqueda indica la dirección del primer elemento.

La estructura interna es muy similar, como se muestra en la Figura 6.

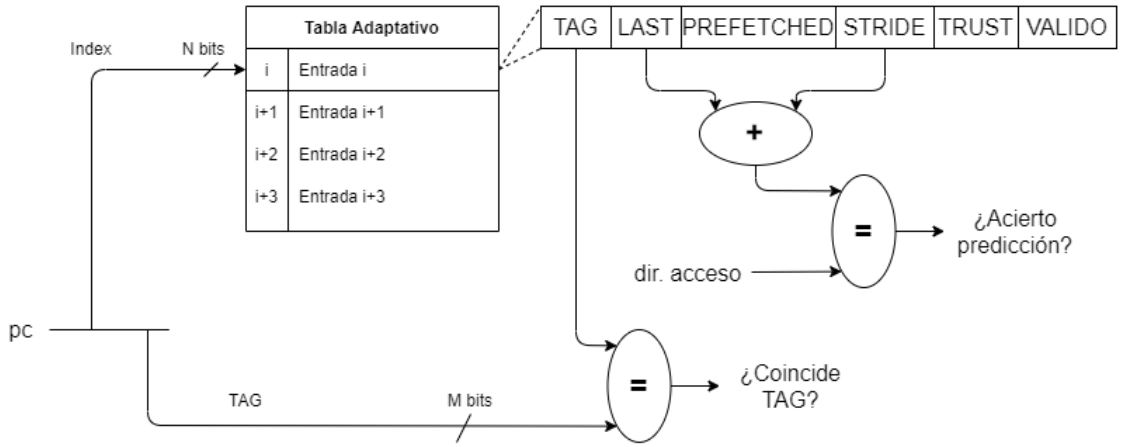

Figura 6: Estructura interna del prebuscador adaptativo

Nuevo campo:

- Prefetched: Contiene la dirección del último elemento prebuscado en el patrón. Permite tener el control de los ya prebuscados.

En cuanto al funcionamiento del prebuscador, del que se proporciona un ejemplo en la sección C.5, el concepto es el mismo que en stride, salvo con algunos matices. Cuando se incremente la confianza en el patrón, se prebusca un número de elementos acorde a la confianza. La cantidad de elementos en cada estado de confianza es arbitraria pero debe ser razonable. El número de elementos prebuscados en función de la confianza sigue la serie de potencias de 2: 0, 1, 2, 4 y 8, lo cual facilita su implementación. Asimismo, tampoco se quiere avanzar en exceso en la prebúsqueda respecto a los elementos que demandan las instrucciones, puesto que podrían expulsar los bloques en cache que van a ser usados en los siguientes accesos (polución). Por ello, es necesario establecer un límite de elementos prebuscados respecto a la demanda. Para lo cual se hace uso del campo *prefetched* que permite calcular el número de elementos que se lleva de ventaja. Cuando este límite se sobrepasa, se limita la prebúsqueda a un solo elemento, independientemente de la confianza. Otro uso de *prefetched* es que permite continuar la prebúsqueda donde se había dejado, independientemente del elemento al que se acceda por demanda.

Este prebuscador es ligeramente más costoso de implementar que el prebuscador stride, pero es mucho más agresivo y permite prebuscar más elementos y con más antelación. Mantiene la desventaja de ser incapaz de detectar patrones irregulares.

3. Simulación

La simulación es una parte fundamental de todo diseño, permite comprobar el correcto funcionamiento del modelo y su integración con el sistema. También es necesario simular para estudiar su viabilidad de implementación, es decir, si se obtiene algún tipo de ganancia con este diseño y si el impacto del mismo compensa el esfuerzo de integración y fabricación.

Para ello, se ha utilizado el entorno de prácticas usado en la asignatura de *Procesadores Comerciales*, impartida en el segundo cuatrimestre de tercer curso en el grado de Ingeniería Informática de la Universidad de Zaragoza. Este entorno se trata de un simulador temporal de la micro-arquitectura de un procesador escrito en lenguaje C.

3.1. Generador de traza Shade

En primera instancia, este simulador se construye sobre la aplicación ***Shade***. Shade es un simulador de arquitectura y generador de traza en el cual se ejecutan y analizan aplicaciones a través de sus ficheros ejecutables [23]. La traza de ejecución obtenida con Shade es idéntica a la que se obtendría de ejecutar el programa en la máquina real. Esta traza contiene no solo las instrucciones ejecutadas, sino además con sus operandos fuente y destino, direcciones efectivas y decisiones de salto, etc. Sobre este sistema, el usuario programa su analizador. Por ejemplo, un analizador puede contar el número de saltos tomados en la ejecución del programa [23], [24].

Otra opción para el analizador, es la adoptada en las prácticas de la asignatura. En lugar de contadores sencillos sobre la traza, se crea un entorno que simula el camino de datos de un procesador (a nivel de micro-arquitectura), al cual se le proporcionan las instrucciones generadas por la traza. En este procesador virtual simula el comportamiento temporal de las instrucciones por todas las etapas del camino de datos, con detenciones, riesgos de datos, estructurales, fallos en cache, saltos, etc.

El despliegue de este entorno se ubica en una máquina de la arquitectura Sparc, *ccia3.unizar.es*, de la Universidad de Zaragoza, puesto que Shade es capaz de emular instrucciones y generar la traza a partir de aplicaciones compiladas para dicha arquitectura. Sobre esta máquina se han realizado todas las pruebas de funcionamiento y depuración del simulador, con pequeñas aplicaciones de comprobación. Sin embargo, esta máquina carece de la potencia de cómputo suficiente para ejecutar, en un tiempo razonable, los *benchmarks* que se pretenden analizar, por lo que es necesario otro entorno.

3.2. Generador de traza Mambo

Una vez comprobado el correcto funcionamiento del simulador en la máquina Sparc, se ha migrado el entorno a otra máquina más potente, *pilgor.cps.unizar.es*, la cual pertenece al *Departamento de Informática e Ingeniería de Sistemas*. Esta máquina es un servidor Huawei TaiShan 2280[25], compuesta por dos procesadores ARM Kunpeng 920 de 48 núcleos cada uno y 320 GB de RAM, entre otras características. Como se ha explicado con anterioridad, Shade es capaz de generar traza a partir de binarios compilados para Sparc, no Arm. Por lo que ha sido necesario usar otra herramienta. Por fortuna, las prácticas de la asignatura Procesadores Comerciales de este año se han realizado en esta máquina, utilizando como generador de traza **MAMBO** [26]. "MAMBO es una herramienta de modificación binaria dinámica de bajo sobrecoste"[26]. "La modificación

binaria dinámica es una técnica para modificar aplicaciones de forma transparente mientras se ejecutan, trabajando a nivel del código máquina nativo[26]. Es decir, es en esta modificación del código donde se genera la traza que se pretende utilizar. Además, a esta herramienta se le pueden añadir *plugins* de forma similar al analizador de Shade, de esta forma se conecta al procesador virtual a simular. Sin embargo, se necesitaron algunas modificaciones para adaptar la lógica del procesador a la arquitectura ARM respecto a Sparc, más de un operando destino, saltos no retardados, etc.

3.3. Procesador virtual inicial

El punto de partida del entorno es un procesador con lanzamiento en orden, con tres caminos de ejecución (aritmética, punto flotante y memoria), *scoreboard* [13], con memoria ideal (todos los accesos son acierto en cache) y terminación en desorden (con *re-order buffer* [13]).

Desde ese punto, se ha modificado el modelo de procesador para que sea similar a Sargantana y se han añadido las características necesarias para poder evaluar el comportamiento y prestaciones de los modelos de prebúsqueda propuestos. Algunas de las modificaciones son: una jerarquía de memorias cache configurables con distintas políticas de reemplazo, caches no bloqueantes con *hit-under-miss* (ver Anexo A), la implementación de distintos modelos de prebuscadores, predictores de salto y métricas de rendimiento, etc.

3.4. Benchmarks

La comprobación y depuración del comportamiento del simulador se puede realizar con aplicaciones de prueba relativamente pequeñas y sencillas, no obstante, nunca se sabe a ciencia cierta si el simulador está libre de *bugs* o errores. Por otra parte, para obtener métricas de rendimiento no sirve cualquier aplicación, y existen una serie de reglas para definir un buen *benchmark*. "La carga de trabajo debe estar estrictamente definida, debe ser reproducible, puede ejecutarse en sistemas variados, las métricas que se obtengan deben ser comparables, debe comprobarse la corrección de la ejecución y debe tener reglas claras sobre la ejecución de la misma"[6]. Además, es recomendable que las métricas de rendimiento se tomen a partir de aplicaciones relativamente grandes y de interés en distintos campos científicos e informáticos. Estas son las características que recogen los *benchmarks* de SPEC [27].

Concretamente, se pretendía usar el conjunto de aplicaciones SPEC CPU 2017[6], que está ideado para medir el rendimiento del procesador. SPEC CPU 2017 está compuesto por varios subconjuntos: *SPECrate 2017 Integer*, *SPECrate 2017 Floating Point*, *SPECspeed 2017 Integer* y *SPECspeed 2017 Floating Point*, de los cuales se iban a utilizar los dos primeros ya que los dos últimos están pensados para evaluar multiprocesadores (ver Anexo A). Los *benchmarks* se hubiesen ejecutado siguiendo las pautas indicadas en *SPECCPU2017 Command Lines* [28], de la Universidad de Arizona. Estas aplicaciones se describen en el Anexo D.

Durante el desarrollo del TFG se realizó una actualización del sistema operativo en *pilgor.cps.unizar.es* que provocó la pérdida de compatibilidad con Mambo. Como consecuencia, gran parte de las aplicaciones SPEC se abortan nada mas iniciarse. En el momento de presentar esta memoria todavía persiste el problema, por lo que solo ha sido posible analizar las prestaciones de los prebuscadores con un pequeño subconjunto de 5 aplicaciones SPEC: 523.xalancbmk_r, 531.deepsjeng_r, 541.leela_r, 557.xz_r y 549.fotonik3d_r.

Con el objetivo de ampliar los resultados obtenidos, se han añadido dos aplicaciones utilizadas en la depuración del simulador. Estas aplicaciones son *matrizc* y *matrizf* las cuales representan un caso muy común, las operaciones con matrices. Concretamente, estas aplicaciones consisten en leer los datos de una matriz, operar con esos datos y almacenarlos en otra matriz del mismo tamaño. El recorrido en el que se leen y almacenan los datos es por columnas y por filas para *matrizc* y *matrizf*, respectivamente. Estas aplicaciones han sido adaptadas aumentando el tamaño de las matrices para que tenga sentido su comparación con el resto de *benchmarks*. Por último, en el momento en el que se solucione dicho problema, la ejecución de los *benchmarks* está totalmente automatizada mediante *scripts* en Bash.

3.5. Configuraciones de estudio

Al margen de las aplicaciones utilizadas para la obtención de métricas de rendimiento, es necesario precisar de qué configuraciones del procesador simulado se quieren extraer dichas métricas, para poder realizar comparaciones. En todas ellas se usa un modelo de procesador similar a Sargantana. Se proponen 10 configuraciones distintas:

- ***NP***: Configuración de referencia. No incorpora ningún mecanismo de prebúsqueda.
- ***ISc***: Sistema con prebúsqueda de instrucciones secuencial. Sin prebúsqueda de datos.
- ***IG***: Sistema con prebúsqueda de instrucciones guiada por predictor de salto. Sin prebúsqueda de datos.
- ***DSc***: Sistema con prebúsqueda de datos secuencial. Sin prebúsqueda de instrucciones.
- ***DSt***: Sistema con prebúsqueda de datos stride. Sin prebúsqueda de instrucciones.
- ***Dad***: Sistema con prebúsqueda de datos adaptativa. Sin prebúsqueda de instrucciones.
- ***AdL***: Misma configuración que en el caso anterior pero con un número reducido de entradas en la tabla del prebuscador adaptativo.
- ***AdM***: Configuración idéntica a la anterior con política de inserción *On Miss* en la tabla del prebuscador adaptativo (ver Anexo E).
- ***I+D***: Sistema con prebúsqueda de instrucciones guiada por predictor y prebúsqueda de datos adaptativa.
- ***LRU***: configuración anterior sustituyendo la política de reemplazo en memoria cache por LRU (*least recent used*) [13].

El objetivo de estas configuraciones es tener unas métricas de referencia de Sargantana para poder estudiar el impacto en rendimiento que tendrían la integración de distintos prebuscadores tanto de datos como de instrucciones. Con ello, se pretende demostrar cuales funcionan mejor en términos de velocidad y coste. Además en el caso del prebuscador adaptativo, se propone el estudio del impacto que tiene reducir el número de entradas de la tabla del prebuscador adaptativo (AaL) y si modificando la política de inserción en dicha tabla se consigue mitigar el impacto de la reducción (AdM). No obstante, en los resultados obtenidos han quedado fuera de análisis.

Por último, las dos configuraciones finales intentan descubrir el rendimiento máximo que puede ofrecer Sargantana con los, a priori, mejores prebuscadores (adaptativo y guiado) con dos políticas de reemplazo en cache diferentes.

3.6. Resultados

En esta sección se van a discutir los resultados obtenidos en las aplicaciones ejecutadas. A continuación, en la Tabla 1, se muestran algunas métricas generales obtenidas de las aplicaciones, con una configuración del procesador sin prebúsqueda, a modo de referencia.

	CPI	Tasa de fallo L1i	Tasa de fallo L1d
523_xalancbmk_r	5.21	2.00 %	10.10 %
531_deepsjeng_r	1.83	1.24 %	3.76 %
541_leela_r	1.34	0.10 %	0.70 %
549_fotonik3d_r	2.93	0.53 %	5.60 %
557_xz_r	1.26	0.10 %	1.57 %
matrizf	3.36	0.10 %	6.20 %
matrizc	33.33	0.10 %	99.99 %

Tabla 1: Resumen de las principales métricas de rendimiento obtenidas para los *benchmarks* sin prebúsqueda.

Como puede observarse, en algunas aplicaciones el CPI (ver Anexo A) es bueno para un procesador en orden, y está cercano al CPI deseable, CPI=1. En estos casos, se corrobora con la tasa de fallo en cache, que los accesos a memoria son casi siempre aciertos. En el Anexo I, se proporcionan resultados completos de la prebúsqueda para todas las aplicaciones ejecutadas. Las aplicaciones que se van a analizar son 523_xalancbmk_r, matrizf y matrizc, al ser los casos más interesantes.

3.6.1. Análisis de 523_xalancbmk_r

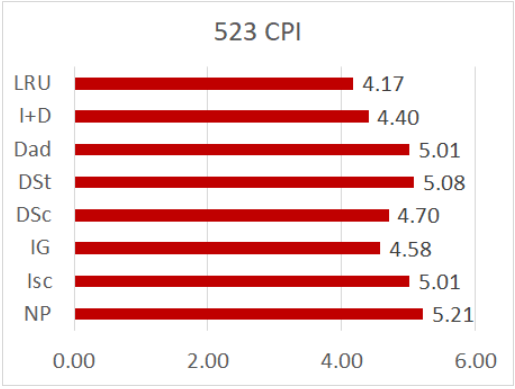

Figura 7

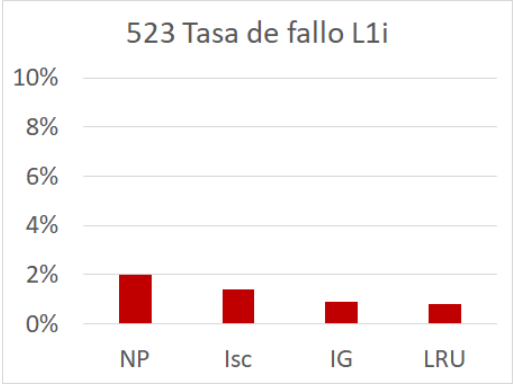

Figura 8

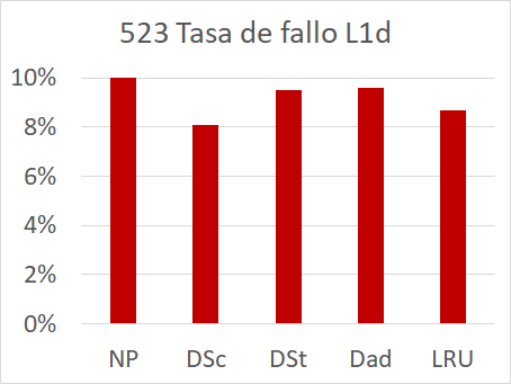

Figura 9

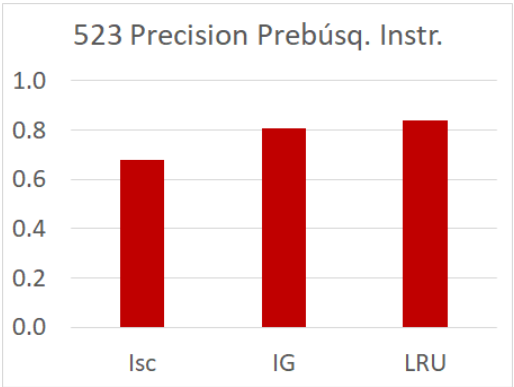

Figura 10

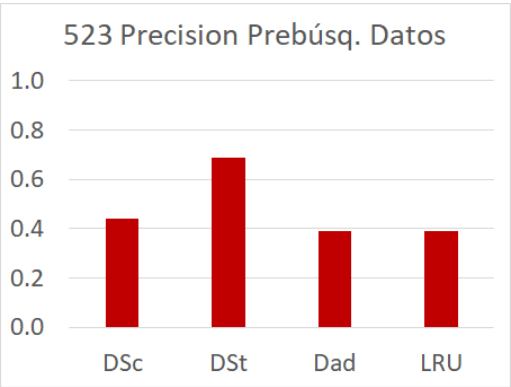

Figura 11

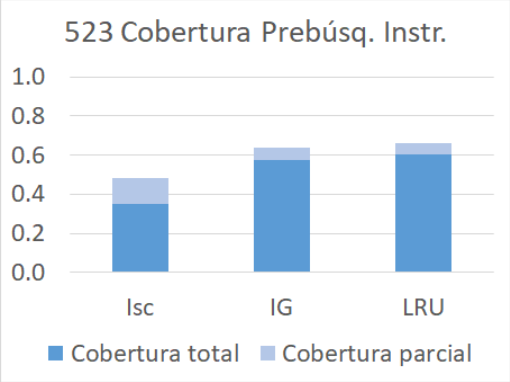

Figura 12

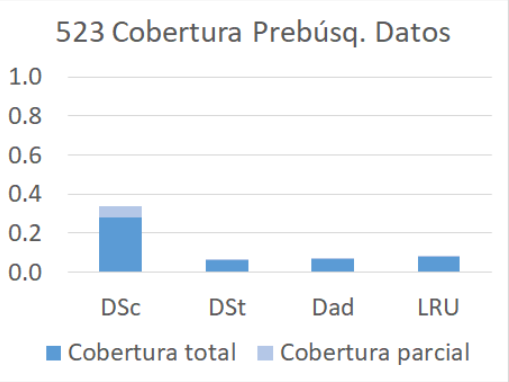

Figura 13

La tasa de fallo en L1i para *523.xalancbmk_r* es del 2 %, por lo que el margen de mejora es escaso. La prebúsqueda de instrucciones funciona correctamente, especialmente el prebuscador guiado por predictor de saltos que logra una cobertura de los fallos entorno al 65 %. Además, gran parte de la cobertura es total (ver Anexo A). Pese a la baja tasa de fallo sin prebúsqueda, la prebúsqueda de instrucciones consigue un *speedup* (ver Anexo A) de 1,14 con reemplazo LRU.

En el acceso a datos, la tasa de fallo en L1d es notablemente mayor, teniendo un margen de mejora alto. Pese a esperarse un *speedup* mayor, el CPI conseguido con la prebúsqueda de datos no refleja esta hipótesis. Para analizar que está ocurriendo, se observa la cobertura y precisión en la prebúsqueda de datos. El mejor rendimiento se obtiene con la prebúsqueda secuencial, con una cobertura del 34 % y un *speedup* de 1.11, no obstante, su precisión no supera el 0.5. Los prebuscadores stride y adaptativo que intentan encontrar patrones fijos tienen un peor rendimiento. Esto es indicativo de que los patrones de acceso son irregulares y no están diseñados para detectarlos. Siendo la tasa de fallo en L1d del 10 %, se consigue un *speedup* limitado a 1,11 en el mejor de los casos. Por último, combinando el prebuscador de datos adaptativo y el prebuscador de instrucciones guiado, junto a la política de reemplazo (LRU), se consigue un *speedup* de 1,25.

3.6.2. Análisis de matrizf

En matrizf se tiene una tasa de fallo en la cache de instrucciones muy baja, como se ha mostrado en la Tabla 1. Esto se debe a que el código del programa es muy pequeño y siempre se están ejecutando las misma instrucciones en bucle, ya presentes en cache. Debido a esto, no se va a analizar la prebúsqueda de instrucciones para esta aplicación, pues el márgen de mejora es ínfimo.

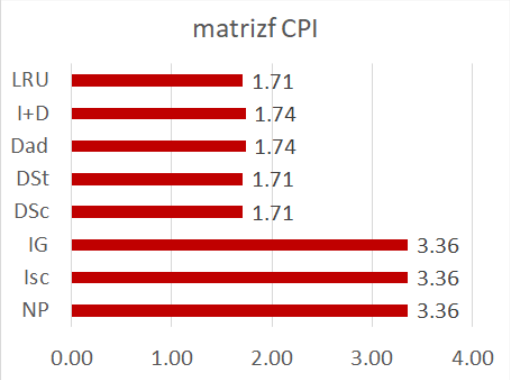

Figura 14

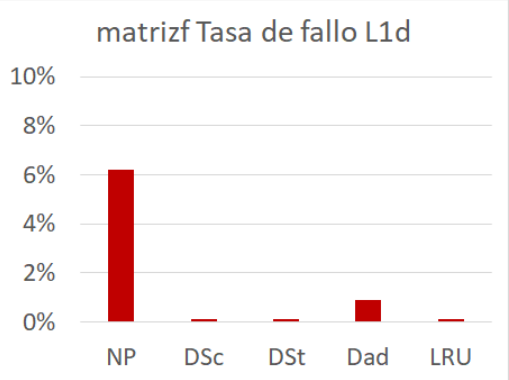

Figura 15

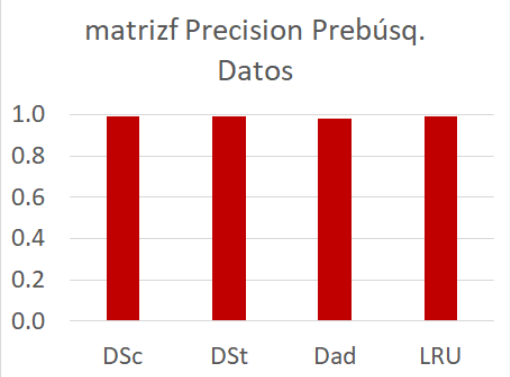

Figura 16

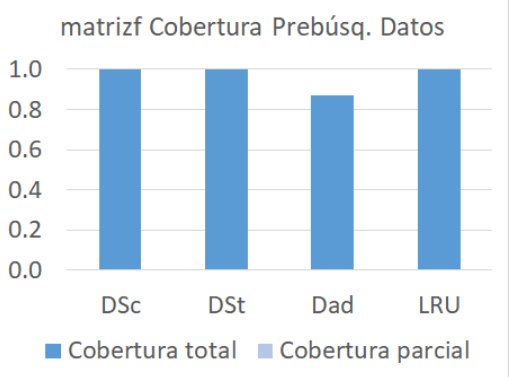

Figura 17

Se recuerda que matrizf recorre las matrices por filas, manera en la que se almacenan sus elementos en memoria. Por ello, todos los mecanismos de prebúsqueda son capaces de traer los bloques de memoria con antelación, puesto que es un recorrido secuencial. Todos ellos tienen una cobertura y precisión alta, lo que reduce la tasa de fallo en cache prácticamente al mínimo. Merece la pena mencionar que en esta aplicación el prebuscador adaptativo se ve perjudicado por el mecanismo de reemplazo aleatorio (utilizado por defecto). Este deterioro en las prestaciones del prebuscador se debe en parte a su agresividad. Como se traen varios bloques a las caches y el reemplazo es aleatorio, cabe la posibilidad que se expulse el bloque que se iba a utilizar a continuación. Al cambiar el algoritmo de reemplazo por LRU, observamos que alcanza el rendimiento de los demás prebuscadores. El *speedup* conseguido es de 1.97, es decir, se ha reducido el tiempo de ejecución de la aplicación a prácticamente la mitad.

3.6.3. Análisis de matrizc

De forma análoga al análisis de matrizf, no se va a comentar la prebúsqueda de instrucciones por el mínimo margen de mejora en el acceso de instrucciones. Como se ha explicado antes, esto se debe al bajo número de instrucciones del código.

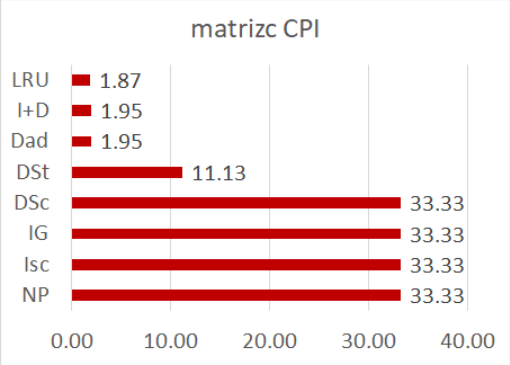

Figura 18

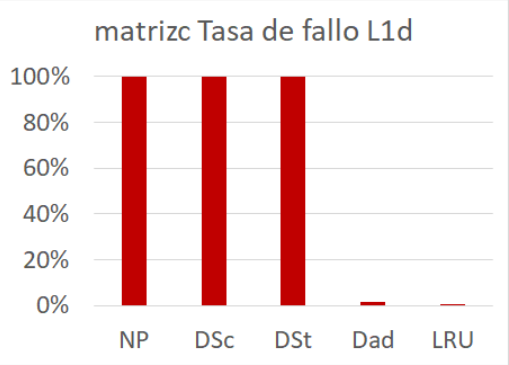

Figura 19

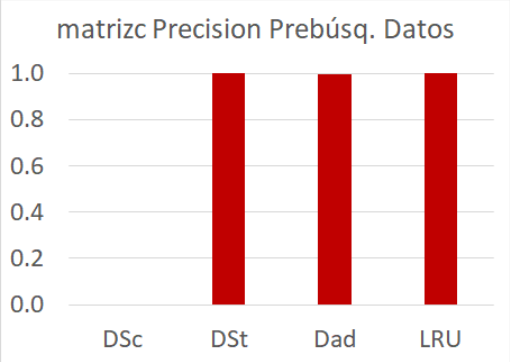

Figura 20

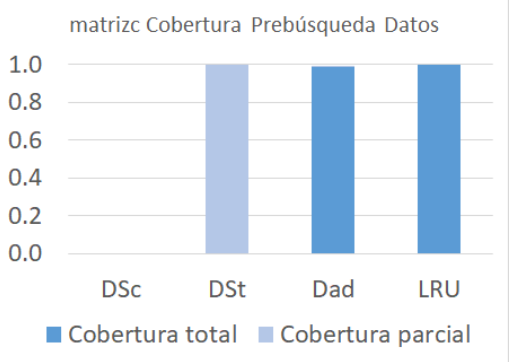

Figura 21

En primer lugar, llama la atención que la tasa de fallos en L1d es cercana al 100 % sin prebúsqueda (además de con prebúsqueda secuencial y con prebúsqueda stride). Esto se debe a que el número de elementos en una columna multiplicado por el tamaño de bloque es mayor que el tamaño de la cache de datos, esto provoca que los bloques en cache que contienen los elementos de la siguiente columna sean expulsados antes de ser utilizados para hacer sitio a la columna actual. En esta aplicación, el número de elementos por columna es 25.000, muy superior a los 16kB de capacidad en L1d, por lo que esta condición se cumple. Por otra parte, en esta aplicación existe una clara diferenciación en el rendimiento de los distintos prebuscadores de datos. La distancia entre las direcciones de un elemento y el siguiente del recorrido por columnas es fija, pero en este caso existe más de un bloque de distancia. Por ello, el prebuscador secuencial está constantemente trayendo bloques innecesarios a cache. En consecuencia, no se obtiene ninguna ventaja con este prebuscador, y aunque no es el caso, incluso podría perjudicar el rendimiento del sistema si las expulsiones

producidas por los bloques prebuscados fuesen bloques útiles. Por otro lado, el prebuscador stride es capaz de detectar el patrón fijo, obteniendo una importante reducción en el CPI. Sin embargo, la tasa de acierto en L1d sigue siendo muy baja, ya que prácticamente toda la cobertura de la prebúsqueda es parcial (ver Anexo A). Esto se debe a que, pese a haber detectado el patrón y haberse prebuscado el siguiente elemento, cuando éste es solicitado por el procesador, todavía se está transfiriendo a L1d. Se han ahorrado los ciclos desde su prebúsqueda hasta su solicitud, pero no se ha evitado el fallo. Esto pone de manifiesto la importancia de la agresividad que añade el prebuscador adaptativo que, como se observa, funciona a la perfección en esta aplicación. Con este último prebuscador se obtiene un *speedup* 17,82 con LRU.

3.7. Conclusiones de simulación

Como conclusión del análisis de los resultados obtenidos, se identifica que la prebúsqueda de instrucciones guiada por predictor de saltos y la prebúsqueda de datos adaptativa son los prebuscadores que mejor se comportan. Esto se debe a que han obtenido los mejores resultados en la mayoría de los casos y en algunos de ellos por diferencias sustanciales. Por otro lado, la política de reemplazo LRU tiene un impacto positivo en el rendimiento del procesador con prebúsqueda, por lo que puede merecer la pena su implementación en Sargantana.

4. Implementación e integración

Una vez concluida la fase de simulación, si se constituye que la prebúsqueda supone un incremento en el rendimiento a un coste asumible, se pasaría en la fase de implementación e integración. En este trabajo, se ha concluido tras los resultados que los mejores prebuscadores de datos y de instrucciones son: prebuscador adaptativo y prebuscador guiado por predictor de saltos, respectivamente. Por limitación de tiempo, se ha trabajado únicamente en la implementación de la prebúsqueda de instrucciones guiada por predictor y su integración con Sargantana, en la cual estaban particularmente interesados desde el CNS. La idea del CNS es incluir este prebuscador de instrucciones en el próximo chip Sargantana que se fabrique.

4.1. Implementación

Se ha optado por una implementación modular del prebuscador de instrucciones por varias razones: minimizar las modificaciones necesarias en la etapa de búsqueda de Sargantana, y permitir una depuración independiente del módulo de prebúsqueda. No obstante, es importante añadir que se trata de un procesador relativamente complejo con capacidad para ejecutar instrucciones de la arquitectura RISC-V, por lo que en su diseño se incluye el tratamiento de excepciones, ejecución de instrucciones atómicas, así como otras características no triviales.

La micro-arquitectura de Sargantana requiere inevitablemente modificar el diseño del prebuscador guiado por predictor de saltos incluido en el módulo de prebúsqueda que se pretende acoplar. Sargantana tiene una etapa de búsqueda segmentada en dos ciclos (la prebúsqueda se realiza en el primero de ellos), mientras que en el modelo teórico del prebuscador se ha considerado de un ciclo. Además, la interfaz con la memoria de instrucciones tiene un único puerto. Por otro lado, es necesario que el módulo de prebúsqueda envíe señales adicionales para el control de la etapa. El diseño final del módulo de prebúsqueda se presenta en la Figura 22, donde se omite la lógica tras señales de salida más simples como *pc*, *petición por demanda* y *petición válida*:

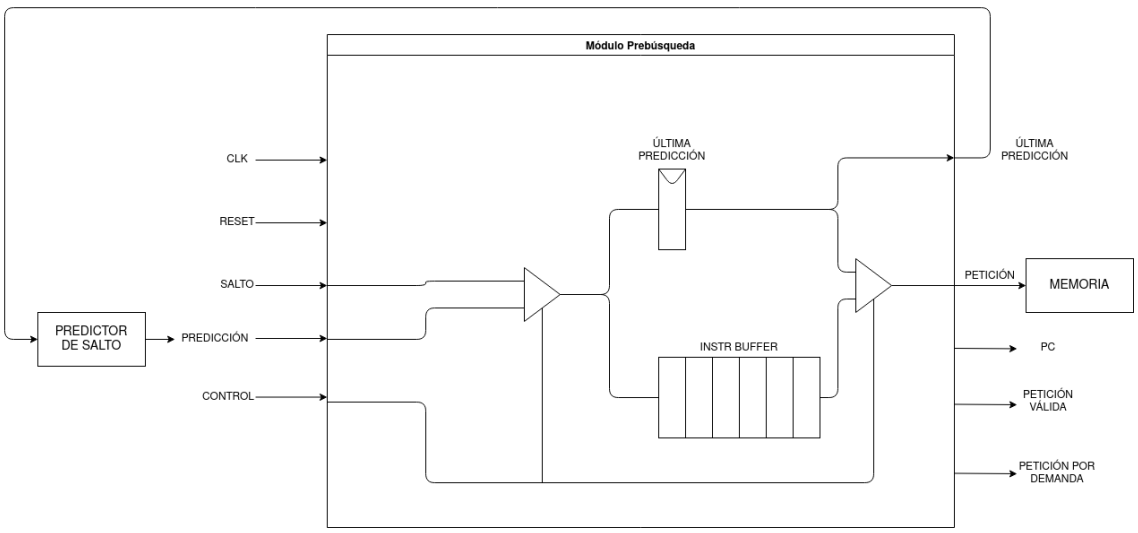

Figura 22: Estructura interna del módulo de prebúsqueda diseñado para Sargantana

Las principales novedades en la prebúsqueda guiada por predictor de saltos implementada, respecto al modelo teórico, residen en la señal de entrada *control*. La señal *control* proviene de la unidad de control, y su función es gobernar los multiplexores, indicando si hay que realizar una petición por demanda, por prebúsqueda o ha habido un fallo en la predicción de salto en ese ciclo. En el caso teórico, se prebuscaba en el ciclo siguiente de detectar un fallo en cache por demanda. Esto está directamente relacionado con la segmentación en dos ciclos de la etapa de búsqueda en Sargantana. Cuando se notifica que existe un fallo en cache en una petición por demanda, realmente se está indicando que la petición que se lanzó en el ciclo anterior es la que ha fallado, por lo que hay que reaccionar en el mismo ciclo en el que se detecta para tener el mismo comportamiento. Por último, se han necesitado algunas señales de salida adicionales para la unidad de control (*petición válida* y *petición por demanda*).

El módulo de prebúsqueda se ha implementado en **RTL** (ver Anexo A). Para ello, se ha especificado en el lenguaje de descripción de hardware VHDL (*Very High Speed Integrated Circuit (VHSIC) Hardware Description Language*). VHDL no debe entenderse como un lenguaje de programación, su utilización consiste en describir el comportamiento lógico de un módulo o entidad de forma textual, donde se especifican los componentes internos utilizados, como se conectan las señales, y los valores que toman las señales de salida en función de las entradas y señales internas del módulo. VHDL y Verilog (un lenguaje similar a VHDL en el que está descrito Sargantana) surgieron en los años 80 como alternativas a aplicaciones de diseño hardware en las que se dibujaban los componentes y sus conexiones de forma explícita, los cuales eran caros y estaban poco estandarizados [7]. En el Anexo F puede encontrarse un ejemplo de especificación gráfica y especificación textual en VHDL.

Una vez descrito el módulo en VHDL, se ha depurado y comprobado su comportamiento mediante bancos de prueba, utilizando la herramienta de verificación y simulación para diseños en VHDL y Verilog, Modelsim PE 10.5 [29]. Con ello, se ha conseguido un módulo de prebúsqueda independiente, correcto y sintetizable. VHDL es un lenguaje independiente de implementación, por lo que una vez comprobado su funcionamiento, es posible tanto su síntesis en una FPGA (placas programables mediante la conexión de bloques lógicos), como su fabricación en una fundición de silicio (ASIC) [7].

4.1.1. Síntesis en FPGA

La síntesis es el proceso que permite pasar de un circuito digital descrito en un lenguaje de especificación a un diseño físico. Se ha realizado una síntesis del diseño del módulo de prebúsqueda en una FPGA Virtex-5 de Xilinx 5vlx50tff1136-1 con el objetivo de analizar su coste en silicio (registros, sumadores, restadores, multiplexores, etc.), y la velocidad a la que puede funcionar correctamente el circuito (frecuencia).

La implementación de este módulo ha constado de tres versiones. En primera instancia, se implementó el módulo reutilizando una cola FIFO implementada en *vhdlguru* para la estructura INSTR BUFFER [30]. Esta implementación funcionaba bien en la simulación, sin embargo, no era eficiente. Por ejemplo, se usaban variables de 32 bits, dónde no eran necesarios tantos bits, por lo que el diseño debía incluir sumadores y restadores de 32 bits, con los retardos que ello acarrea. También incluía un multiplexor para seleccionar la salida del *buffer* FIFO. Con todo ello, el retardo obtenido en la síntesis era mucho mayor de lo necesario; unos 170 MHz como mínimo, cuando se usaba una FIFO de sólo cuatro entradas (en la simulación se han utilizado 32).

En una segunda implementación, se optimizó el diseño para mejorar estos resultados. Se diseñó un nuevo *buffer*, ajustando los módulos utilizados a las necesidades del diseño, eliminando lógica secuencial innecesaria, sustituyendo los sumadores y restadores de 32 bits por un contador de 6 bits (los mínimos para contar 32 elementos), y fijando que la cabeza del *buffer* estuviese siempre en el registro 0. Con estos cambios la versión del *buffer* con cuatro registros pasó a 470MHz, prácticamente la velocidad máxima que se podía alcanzar.

La tercera implementación ha consistido en modificar el diseño existente para que utilice parámetros genéricos además de reducir el *hardware* necesario. Estos parámetros permiten que un mismo módulo pueda instanciarse con distintos tamaños. Así se puede cambiar el tamaño del *buffer* sin necesidad de alterar su código. La utilización de genéricos complica bastante la fase de diseño del módulo, y lo hace algo menos legible. Sin embargo, es muy útil cuando no se sabe qué tamaño se va a utilizar, dado que permite explorar el espacio de diseño y hacer pruebas con distintos tamaños sin necesidad de reescribir ningún módulo. En el diseño final se han utilizado dos parámetros genéricos: el tamaño de los registros, que en esta arquitectura es 64, pero que podría ser 32 en otras arquitecturas, y el número de registros que incluye el *buffer*, que puede ser cualquier potencia de 2. Para hacer este diseño con genéricos, no es suficiente con instanciar un número de registros variable y de tamaño genérico. También se han tenido que adaptar los módulos de decodificación, multiplexores y contadores para que sean capaces de trabajar con un número genérico de elementos de entrada. En el Anexo G se ilustra el uso de genéricos mediante un sencillo ejemplo.

A continuación se van a comentar brevemente los resultados de síntesis en la implementación final. En el Anexo H se encuentra el informe completo. Con 4 entradas en el *buffer*, el primer diseño puede ir a una velocidad máxima de 170Mhz, el segundo alcanza una velocidad de 470Mhz, mientras que el tercer diseño logra funcionar a 434MHz. Este último es algo más lento que el segundo, pese a ser un diseño más maduro, debido a que se hace un uso más eficiente del *hardware*, utilizando un 31 % menos (ver Anexo H). Además, es una frecuencia suficientemente buena teniendo en cuenta que la FPGA puede alcanzar un máximo de 500Mhz. Tras dar el diseño como válido, se sintetizó el modelo grande, con 32 entradas en el *buffer*, para asemejarse al de la simulación. Esto tiene un impacto en la velocidad, puesto que los multiplexores, sumadores, restadores, decodificadores, etc. son más grandes y como consecuencia, más lentos. Las frecuencias alcanzadas son 311Mhz y 281MHz para el segundo y tercer diseño, respectivamente.

Por otro lado, el número de *slices* (ver Anexo A) es del 20 % en el segundo diseño, y únicamente 4 % en el diseño final, lo cual demuestra que es un diseño más eficiente y consume pocos recursos de la FPGA. Por último, la frecuencia final, 281 MHz, supone un incremento importante respecto a los 170 MHz del primer diseño, teniendo en cuenta que el *buffer* pasó de 4 a 32 entradas. El tiempo de ciclo del módulo de prebúsqueda en este diseño no supone un cuello de botella en el rendimiento del sistema, puesto que se está manejando una frecuencia de 100MHz en FPGA para Sargantana [1].

[1] 100MHz para un procesador como Sargantana pueden parecer muy lentos, pero es una buena velocidad para un diseño complejo en FPGA. Las FPGAs permiten implementar cualquier diseño de forma rápida, sin necesidad de fabricar un nuevo circuito integrado, dado que disponen de bloques configurables versátiles que se pueden utilizar para conseguir la funcionalidad deseada. Pero los retardos de esos bloques son mucho mayores que los de se obtienen en los diseños implementados en un circuito integrado a medida.

4.2. Integración con Sargantana

El paso final es la integración del módulo de prebúsqueda en Sargantana. Pese a ser un diseño modular, el prebuscador guiado por predictor de saltos sustituye componentes básicos de la etapa de búsqueda como el registro PC (por un *buffer*) por lo que se modificado gran parte de la etapa de búsqueda original. También se necesita modificar la comunicación con la cache de instrucción pues, mientras se espera a la respuesta de la cache a una petición por demanda, se están realizando otras por prebúsqueda, y es necesario distinguir las respuestas para decidir cuando se puede continuar con la ejecución de la instrucción. Estas modificaciones detallan a grandes rasgos en qué se ha trabajado para integrar el módulo en Sargantana, aunque se han invertido tiempo en otros aspectos, como el tratamiento de excepciones, la propagación de señales y la interpretación de señales de la unidad de control (UC). Pese a los esfuerzos realizados, por falta de tiempo y la complejidad encontrada, no se ha logrado verificar la integración completa con Sargantana. Si bien, sí que se ha podido verificar que los módulos desarrollados cumplen exactamente la especificación inicial a través de la simulación de bancos de pruebas. Sin embargo, estás mismas pruebas no se han podido realizar en el procesador completo, por lo que se deja como trabajo futuro junto a la prebúsqueda de datos.

En la Tabla 2 se indican casos comprobados mediante bancos de prueba en *Modelsim*. Se han comprobado que las salidas son correctas para cada caso de prueba.

Caso de prueba	Funciona correctamente
Petición por demanda	Sí
Fallo en cache (petición por prebúsqueda)	Sí
Buffer lleno	Sí
Buffer vacío	Sí
Fallo de predicción	Sí

Tabla 2: Casos de prueba verificados en el módulo de prebúsqueda

5. Conclusiones

El objetivo principal de este trabajo fin de grado era abarcar todas las fases del desarrollo de un prebuscador; desde el estudio de varios modelos teóricos y adquisición de conocimientos en el campo, pasando por la simulación para poder demostrar las ventajas de la prebúsqueda, hasta su especificación e integración en un procesador real. Como se ha ido exponiendo en la presente memoria, este objetivo se ha cumplido en su totalidad, eso sí, con diversos grados de profundidad.

Los logros alcanzados en este trabajo fin de grado merecen ser destacados. Se han estudiado distintos mecanismos de prebúsqueda, los cuales se han integrado en un procesador simulado muy similar a Sargantana, un procesador RISC-V real en el que se pretende acoplar la prebúsqueda. De este simulador, se han obtenido diversas métricas de rendimiento en *benchmarks* SPEC, utilizados globalmente para la comparación de procesadores. Estás métricas han permitido analizar y seleccionar que mecanismos de prebúsqueda se comportarían mejor en el procesador real. Se podrá hacer un análisis mucho más profundo y exhaustivo en el momento que se recupere la compatibilidad entre Mambo y el sistema operativo de *pilgor.cps.unizar.es*, puesto la ejecución de los *benchmarks* SPEC está automatizada. Tras esto, se ha implementado un diseño modular y configurable de la prebúsqueda de instrucciones en RTL. Este diseño se ha verificado y optimizado mediante síntesis para que el coste de implementación sea lo más bajo posible. Por último, se ha iniciado la integración de este módulo en el diseño de Sargantana.

Aquellos objetivos que no han podido completarse por problemas tecnológicos o de tiempo se dejan como propuesta de trabajo futuro. En la simulación, queda pendiente obtener las métricas de rendimiento en todos los *benchmarks* SPEC, cuando la compatibilidad con la herramienta Mambo lo permita. A partir de estos datos, obtener los tamaños óptimos de las estructuras de los prebuscadores en términos de coste y rendimiento. En la parte de integración, se propone terminar la ya iniciada prebúsqueda de instrucciones, en la cual están especialmente interesados desde el CNS. El objetivo del CNS es incluir esta prebúsqueda en el próximo chip que fabriquen. Por otro lado, se plantea implementar la prebúsqueda de datos e integrarla a Sargantana de forma similar.

A modo de retrospectiva, se pondrá de manifiesto el aprendizaje personal. En este proyecto se ha aprendido una metodología, pues los conocimientos adquiridos no se limitan únicamente a los prebuscadores estudiados. Esta metodología es común en el desarrollo de todo procesador. Por otro lado, en este proyecto se han asentado y profundizado multitud de conceptos vistos en asignaturas del grado; como la jerarquía de memorias cache, la predicción de saltos, la prebúsqueda, el diseño RTL, así como la importancia de la simulación, automatización y el diseño modular, entre otros. Se ha adquirido una mayor destreza en el uso de herramientas como Mambo, Modelsim o en el lenguaje VHDL, y se ha trabajado con otras nuevas como los *benchmarks* de SPEC o el lenguaje de descripción hardware Verilog. Además, se ha comprobado la complejidad subyacente en el desarrollo de un procesador real como Sargantana, que carece de las simplificaciones de modelos pedagógicos. En este desarrollo se han puesto en práctica todos los conocimientos en arquitectura y organización adquiridos durante el grado.

Se concluye que, pese a las dificultades encontradas, se ha logrado cumplir con los objetivos del proyecto mientras las herramientas lo han permitido, destacando la visión global adquirida en el desarrollo de procesadores, que sin duda será útil en proyectos futuros. Como culminación, se ha alcanzado un buen nivel de satisfacción personal conforme a las horas invertidas y los avances conseguidos. En el Anexo 23 se aporta un diagrama de Gantt del proyecto.

6. Bibliografía

Referencias

[1] A. J. Smith, «Cache Memories,» *ACM Computing Surveys (CSUR)*, vol. 14, n.º 3, págs. 473-530, 1982, ISSN: 15577341. DOI: 10.1145/356887.356892.

[2] S. P. Vanderwiel y D. J. Lilja, «Data Prefetch Survey,» vol. 32, n.º 2, págs. 174-199, 2001.

[3] *European Processor Initiative*. dirección: https://www.european-processor-initiative.eu/ (visitado 31-08-2021).

[4] *BSC-CNS Barcelona Supercomputing Center - Centro Nacional de Supercomputacion*. dirección: https://www.bsc.es/ (visitado 30-08-2021).

[5] *DRAC project*. dirección: https://drac.bsc.es (visitado 29-08-2021).

[6] *SPEC CPU® 2017 Overview*. dirección: https://www.spec.org/cpu2017/Docs/overview.html (visitado 30-08-2021).

[7] *VHDL*. dirección: https://vhdl.es/ (visitado 11-09-2021).

[8] *SGA1 (Specific Grant Agreement 1) OF THE EUROPEAN PROCESSOR INITIATIVE (EPI)*. dirección: https://cordis.europa.eu/project/id/826647/es (visitado 31-08-2021).

[9] *RISC-V*. dirección: https://riscv.org/ (visitado 31-08-2021).

[10] A. Waterman y K. Asanovic, «The RISC-V Instruction Set Manual Volume I: Unprivileged ISA,» *Tech report*, vol. I, 2019. dirección: https://riscv.org/technical/specifications/.

[11] *Semidynamics - Avispado - High Bandwidth RISC-V IP*. dirección: https://semidynamics.com/products/avispado (visitado 30-08-2021).

[12] *Untethered lowRISC tutorial*. dirección: https://lowrisc.org/docs/untether-v0.2/ (visitado 29-08-2021).

[13] J. L. Hennessy y D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 5.ª ed. Amsterdam: Morgan Kaufmann, 2012, ISBN: 978-0-12-383872-8.

[14] C. Chen, G. Novick y K. Shimano, *What is RISC?* 2000. dirección: https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/whatis/index.html (visitado 29-08-2021).

[15] S. Przybylski, M. Horowitz y J. Hennessy, «Performance Tradeoffs in Cache Design.,» págs. 290-298, 1988, ISSN: 0163-5964. DOI: 10.1145/633625.52433.

[16] Intel, «Intel 64 and IA-32 Architectures Optimization Reference Manual, vol. 1-3,» n.º 253665, 325383, 325384, 2016. dirección: http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html.

[17] ——, «Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes,» *System*, vol. 3, n.º 253665, 2011.

[18] A. Navarro-Torres, J. Alastruey-Benedé, P. Ibáñez-Marín y V. Viñals-Yúfera, «Memory hierarchy characterization of SPEC CPU2006 and SPEC CPU2017 on the Intel Xeon Skylake-SP,» *PLoS ONE*, vol. 14, n.º 8, 2019, ISSN: 19326203. DOI: 10.1371/journal.pone.0220135.

[19] I. Business y M. Corporation, «POWER9 Processor User ' s Manual OpenPOWER,» n.º April, 2018.

[20] G. Reinman, B. Calder y T. Austin, «Fetch directed instruction prefetching,» *Proceedings of the Annual International Symposium on Microarchitecture*, págs. 16-27, 1999, ISSN: 10724451. DOI: 10.1109/micro.1999.809439.

[21] I. C. K. Chen, C. C. Lee y T. N. Mudge, «Instruction prefetching using branch prediction information,» *Proceedings - IEEE International Conference on Computer Design: VLSI in Computers and Processors*, págs. 593-601, 1997. DOI: 10.1109/iccd.1997.628926.

[22] L. M. Ramos, J. L. Briz, P. E. Ibáñez y V. Viñals, «Multi-level adaptive prefetching based on performance gradient tracking,» *Journal of Instruction-Level Parallelism*, vol. 13, 2011.

[23] B. Cmelik y D. Keppel, «Shade: a fast instruction-set simulator for execution profiling,» *Performance Evaluation Review*, vol. 22, n.º 1, págs. 128-137, 1994, ISSN: 01635999. DOI: 10.1145/183019.183032.

[24] P. Lewis, «An Introduction to Shaders,» *Html5Rocks*, págs. 1-11, 2014.

[25] *Servidor TaiShan 2280*. dirección: https://e.huawei.com/es/products/servers/taishan-server/taishan-2280-v2 (visitado 30-08-2021).

[26] A. Rosenthal, «Mambo,» *New Orleans Review*, vol. 31, n.º 2, pág. 82, 2005, ISSN: 00286400. DOI: 10.1145/2896451.

[27] *Standard Performance Evaluation Corporation*. dirección: https://www.spec.org (visitado 30-08-2021).

[28] T. Limaye, Ankur and Adegbija, «SPEC CPU2017 Command Lines,» *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, págs. 149-158, 2018.

[29] M. G. Corporation, «ModelSim ® PE Tutorial,» 2013.

[30] *vhdlguru - Basic model of FIFO Queue in VHDL*. dirección: https://vhdlguru.blogspot.com/2010/03/basic-model-of-fifo-queue-in-vhdl.html (visitado 18-09-2021).

[31] *Wikipedia - RTL*. dirección: https://en.wikipedia.org/wiki/Register-transfer_level (visitado 11-09-2021).

[32] *Compiler Explorer*. dirección: https://godbolt.org/ (visitado 05-09-2021).

[33] C.-b. Prefetching, P. Ib, V. Vifials, L. Briz y M. J. Garzarh, «Characterization and Improvement of Load / Store,» págs. 369-376, 1998.

[34] *Wikimedia - Full-adder.svg*. dirección: https://commons.wikimedia.org/wiki/File:Full-adder.svg (visitado 11-09-2021).

[35] *allaboutfpga - VHDL Code for Full Adder*. dirección: https://allaboutfpga.com/vhdl-code-for-full-adder/#VHDL_Code_for_Full_Adder (visitado 18-09-2021).

A. Glosario

- **Bloques cache**: La comunicación entre las memorias cache y la memoria principal es relativamente lenta, por ello, aprovechando el principio de localidad espacial, es preferible enviar agrupaciones de datos a las caches en lugar de hacerlo individualmente. Estas agrupaciones de varios bytes consecutivos se llaman bloques cache.
- **Localidad espacial**: Propiedad de los accesos a memoria de un procesador. Indica que hay una probabilidad muy alta de que si se accede a un elemento en memoria, se accederá a otro muy cercano en un futuro próximo.
- **Camino de datos**: Es un conjunto de unidades funcionales, registros y conexiones que atraviesan las señales en la ejecución de una instrucción.
- **ISA**: *Instruction Set Architecture*, arquitectura del lenguaje máquina o simplemente arquitectura, define el conjunto de instrucciones que es capaz de ejecutar un procesador, así como sus comportamientos, tamaños y codificaciones.
- **RTL**: *Register Transfer Level*. Abstracción de diseño en circuitos digitales en el que se modela a nivel comunicación de señales entre registros, puertas lógicas, bloques lógicos, etc [31].
- **Computación exaescala**: Sistema capaz de ejecutar 10^{18} FLOPS (operaciones de punto flotante por segundo).
- **Benchmark**: Aplicación utilizada como referencia para medir las prestaciones de un sistema.
- **Multiprocesador**: Máquina compuesta por dos o más procesadores.
- **Slices**: Las FPGA se dividen en agrupaciones de bloques lógicos denominados *slices*. Los *slices* sirven para implementar la lógica de los diseños sintetizados.
- **Fallos Parciales**: Aquellos accesos a memoria que, pese a producir un fallo en cache, el elemento solicitado ya estaba en camino.
- **Cobertura**: Cociente de los fallos solucionados por prebúsqueda entre los fallos que se habrían producido sin prebúsqueda. También puede calcularse como la suma de la cobertura total y la cobertura parcial. Otro nombre con el que se ha definido es cobertura general.
- **Cobertura total**: Cociente de los fallos totalmente solucionados por prebúsqueda entre los fallos que se habrían producido sin prebúsqueda. Por totalmente solucionados se entiende que cuando el procesador solicita el bloque en cache se produce un acierto.
- **Cobertura parcial**: Cociente de los fallos parcialmente solucionados por prebúsqueda entre los fallos que se habrían producido sin prebúsqueda. Se entiende por parcialmente solucionados a aquellos bloques que se están transfiriendo a la memoria cache en el momento que son solicitados por el procesador. Se produce un fallo en cache, pero la penalización es menor.
- **Speedup**: Aceleración obtenida respecto dos ejecuciones de un mismo programa, siendo la ejecución de referencia el numerador.
- **Precisión**: Tasa de bloques prebuscados utilizados respecto a bloques prebuscados traídos.
- **Polución**: Propiedad adversa que se da en las memorias cache cuando se traen bloques innecesarios, provocando la expulsión de otros que sí se van a utilizar en el futuro.
- **CPI**: Ciclos por instrucción.

- **hit-under-miss**: Tipo de memorias caches no bloqueantes, que permiten realizar nuevos accesos a cache tras producirse un solo fallo. Si se produce un segundo, se detiene el uso de memoria hasta que se resuelve el primero. En otras palabras, permiten aciertos consecutivos tras un fallo.

B. Diagrama de Gantt

En este anexo se presenta el diagrama de Gantt de este trabajo fin de grado. Destaca que el desarrollo del simulador ocupó una gran parte del tiempo de este proyecto. Por otro lado, se reflejan los problemas surgidos con los *benchmarks* SPEC en el alargue de la obtención de resultados.

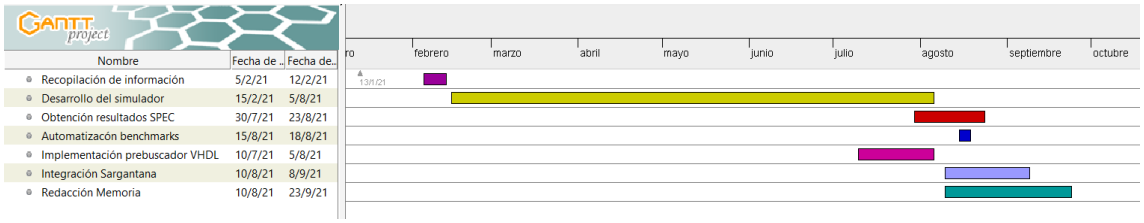

Figura 23: Diagrama de Gantt del proyecto

En cuanto a la estimación en horas, se calcula que se han invertido algo más de 400 horas. Esto se debe a que el proyecto abarca desde febrero a septiembre. Evidentemente, la carga de trabajo no ha sido siempre constante, hasta junio se compaginó con asignaturas del grado. La parte en la que se ha invertido más tiempo es, sin duda, el desarrollo del simulador que ha requerido de cerca de 200 horas por la cantidad de elementos que se han necesitado añadir.

C. Ejemplos prebúsqueda

C.1. Premisas generales de los ejemplos

Para todos los ejemplos de este anexo, se asumen una serie de premisas:

- Para eliminar la confusión que puede presentar trabajar con direcciones de memoria y números hexadecimales, se van a sustituir por números de bloque y números de dato/instrucción. Una dirección será representada por la tupla {número de bloque}.{número de instrucción/-dato dentro del bloque}. La dirección correspondiente al elemento Y perteneciente al bloque X es X.Y.
- Se asume un tamaño de bloque cache de **cuatro** instrucciones/datos, de tal forma que el primer elemento (o elemento 0) del bloque X es X.0 y el último (o elemento 3) es X.3. Además, para representar la dirección del propio bloque X, que es la misma que la del primer elemento, se utilizará la tupla X.0. Por último, la dirección del primer bloque y del primer elemento del espacio de direcciones corresponde a 0.0.
- Por último, recordar que esta abstracción es una simplificación de como operan las memorias caches con direcciones. En realidad, una dirección de memoria (virtual o física) se descompone en tres campos. Los $\log_2(Tamaño\ Bloque)$ bits menos significativos de la dirección para seleccionar el elemento dentro de un bloque, los $\log_2(Número\ Conjuntos\ Cache)$ bits siguientes representan el conjunto de cache al que se asocia el bloque, y los bits restantes de más peso, como bits de etiqueta o *tag*, para distinguir bloques cuyos bits de número de bloque coincidan.

C.2. Ejemplo de funcionamiento del prebuscador OBL

En este ejemplo se representa el funcionamiento del prebuscador *One block lookahead*. Puesto que el funcionamiento es idéntico en instrucciones y datos no se va a hacer distinción alguna. Por otro lado, se siguen las indicaciones de la sección del Anexo C.1

Ciclo	Dirección del bloque solicitado	Dirección del dato/instr.	¿Prebuscar?	Dirección prebúsqueda
1	0.0	0.0	Sí	1.0
2	0.0	0.1	Sí	1.0
3	0.0	0.2	Sí	1.0
4	0.0	0.3	Sí	1.0
5	1.0	1.1	Sí	2.0
6	1.0	1.2	Sí	2.0

Tabla 3: Tabla de ejemplo de funcionamiento del prebuscador OBL

Como puede verse en la tabla 3, el funcionamiento de la prebúsqueda OBL es muy sencillo. Siempre se prebusca la dirección de bloque siguiente a la dirección de bloque accedido. Si se accede a todos los elementos del bloque, se está realizando una petición de prebúsqueda al mismo bloque cuatro veces, lo que supone un uso ineficiente de la cache.

C.3. Ejemplo de funcionamiento del prebuscador guiado por predictor de saltos

En este ejemplo, se va a ilustrar el funcionamiento del prebuscador guiado por predictor de saltos. El ejemplo escogido es un código C, mostrado en el listado 1, el cual se corresponde a una función de inicialización típica. El código ensamblador mostrado en el listado 2 corresponde al código máquina resultante de compilar el código C anterior con GCC 9.3 para la arquitectura ARM[32] (se ha elegido ARM porque es para la que genera traza la aplicación Mambo del simulador).

Además, se presupone una predicción inicial del predictor de saltos no tomado, una latencia fija en caso de fallo en cache de 10 ciclos (contando el ciclo de solicitud), resolución del salto al ciclo siguiente, y se tienen en cuenta las indicaciones explicadas en la sección del Anexo C.1. Convenientemente, la dirección de las instrucciones del código son las iniciales del espacio de direcciones; 0.0, 0.1, 0.2...

```
int vec[10];
int inicializacion = 0;
int cuenta;

void ini(){
    int i = 0;
    while(i < 10){
        vec[i] = 0;
        i++;
    }
    inicializacion = 1;
}
```

Listing 1: Código C de una función de inicialización

```
@Uso de los registros:
@r3: Contiene la dir. de vec antes y durante el bucle
@    posteriormente la dir. de inicializacion
@r1: Contiene la dir. del ultimo elemento de de vec
@r2: Almacena el valor 0 el bucle y luego el valor 1

@ Se indica a la izquierda de las instrucciones su direccion

ini:
 0.0    movw    r3, #:lower16:vec
 0.1    movt    r3, #:upper16:vec
 0.2    add     r1, r3, #40
 0.3    movs    r2, #0
bucle:
 1.0    str     r2, [r3], #4
 1.1    cmp     r3, r1
 1.2    bne     bucle
 1.3    movw    r3, #:lower16:.LANCHOR0
 2.0    movt    r3, #:upper16:.LANCHOR0
 2.1    movs    r2, #1
 2.2    str     r2, [r3]
 2.3    bx      lr
```

Listing 2: Código ARM generado a partir de la función en C del listado 1

Ciclo	Dirección de solicitud a memoria	Acierto en Cache	Fallo predicción	Fuente de solicitud
1	0.0	No	No	Demanda
2	0.1	No importa	No	Prebúsqueda
3	0.2	No importa	No	Prebúsqueda
4	0.3	No importa	No	Prebúsqueda
5	1.0	No importa	No	Prebúsqueda
6	1.1	No importa	No	Prebúsqueda
7	1.2	No importa	No	Prebúsqueda
8	1.3	No importa	No	Prebúsqueda
9	2.0	No importa	No	Prebúsqueda
10	2.1	No importa	No	Prebúsqueda
11	0.1	Sí	No	Demanda
12	0.2	Sí	No	Demanda
13	0.3	Sí	No	Demanda
14	1.0	Sí	No	Demanda
15	1.1	Sí	No	Demanda
16	1.2	Sí	No	Demanda
17	-	-	Sí	-
18	1.0	Sí	No	Demanda
19	1.1	Sí	No	Demanda

Tabla 4: Ejemplo de funcionamiento del prebuscador guiado por predictor de saltos

En el cuadro 4 se muestran las peticiones a memoria cache de instrucciones durante la ejecución del código del listado 2. En el primer ciclo, se pide a la cache la primera instrucción por demanda (porque es la instrucción que hay que ejecutar), sin embargo, la respuesta es que no está disponible, es un fallo en cache, por lo que la fuente de las peticiones cambiará. En el ciclo siguiente, y durante otros ocho más, se lanzan peticiones de prebúsqueda hacia la cache a direcciones indicadas por el predictor de saltos (como solo hay un salto y en esta ejecución se predice no tomado siempre es la instrucción siguiente). Como las peticiones son de prebúsqueda, no importa si aciertan o no en cache, se realizan para que en caso de no estar el bloque almacenado, se pida lo antes posible para que cuando realmente haya que usarlo ya esté presente. En el ciclo diez, la memoria cache notifica que puede proporcionar la instrucción que se había solicitado nueve ciclos antes, por lo que al siguiente ciclo se continuará con la ejecución del programa. Durante el parón anterior, se han prebuscado tres bloques distintos (uno ya se estaba trayendo), por lo que las siguientes instrucciones por demanda están produciendo todo aciertos que sin prebúsqueda no se darían (solo serían aciertos aquellas pertenecientes al mismo bloque de la instrucción que provocó el primer fallo). Por último, y para ilustrar todos los casos del prebuscador, se llega al salto, el cual se había predicho como no tomado y así ha continuado la prebúsqueda. En el ciclo 16, se solicita la dirección correspondiente a esta instrucción, acertando en cache, y se envía a la siguiente etapa (decodificación). Como se asume que los saltos se resuelven con latencia uno, el procesador descubre que su predicción era errónea

en el ciclo 17. Al siguiente ciclo se solicita la dirección efectiva de salto, en este caso la primera instrucción del bucle. Además se vacían los *buffers* del prebuscador porque estaban basados en una predicción errónea.

Como conclusión de este ejemplo, se ilustra cómo en los ciclos que el procesador estaría detenido en etapa de búsqueda o *fetch* esperando una instrucción, se pueden aprovechar para pedir las instrucciones siguientes que indica el predictor de saltos, el cual únicamente puede equivocarse en instrucciones de salto (algo infrecuente, por otra parte [20], [21]).

C.4. Ejemplo de funcionamiento del prebuscador stride

A continuación, se presenta un ejemplo de funcionamiento del prebuscador stride en un caso real. El código C que se ejecuta se muestra en el listado 3, correspondiente a un simple algoritmo de multiplicación de matrices, una situación bastante común en infinidad de aplicaciones, en la que una de las matrices debe ser recorrida por columnas. En el listado 4 se adjunta código ensamblador resultante de su compilación con GCC 9.3 con optimización O2 para la arquitectura ARM [32]. De todas las instrucciones, se va a analizar el comportamiento del prebuscador para el *load* de la línea 19 (recorre la matriz m2 por columnas), correspondiente al bucle interno. Además, se van a seguir las pautas indicadas en la sección C.1 y la dirección del primer dato de la matriz m2 corresponde a la dirección 0.0, por simplificación.

```
1  int m1[8][8];
2  int m2[8][8];
3  int res[8][8];
4
5  void func(){
6      for (int i = 0; i < 8; i++) {
7          for (int j = 0; j < 8; j++) {
8              res[i][j] = 0;
9              for (int k = 0; k < 8; k++) res[i][j] += m1[i][k] * m2[k][j];
10         }
11     }
12 }
```

Listing 3: Código ARM de una función de multiplicación de matrices

```
1  func:
2          ldr     ip, .L10
3          push    {r4, r5, r6, r7, r8, r9, lr}
4          add     r5, ip, #32
5          ldr     r8, .L10+4
6          add     r9, ip, #256
7          ldr     lr, .L10+8
8  .L2:
9          movw    r6, #:lower16:m2
10         movt    r6, #:upper16:m2
11         mov     r7, r8
12 .L4:
13         adds    r7, r7, #4
14         mov     r1, r6
15         mov     r3, ip
16         movs    r2, #0
17 .L3:
18         ldr     r0, [r3, #4]!
19         ldr     r4, [r1], #32
20         cmp     r3, r5
21         mla     r2, r4, r0, r2
```

```
22          bne     .L3
23          adds    r6, r6, #4
24          str     r2, [r7]
25          cmp     r6, lr
26          bne     .L4
27          add     ip, ip, #32
28          add     r5, r3, #32
29          cmp     ip, r9
30          add     r8, r8, #32
31          bne     .L2
32          pop     {r4, r5, r6, r7, r8, r9, pc}
33  .L10:
34          .word   m1-4
35          .word   res-4
36          .word   m2+32
```

Listing 4: Código ARM de una función de multiplicación de matrices

Iteración	Dato accedido (k, j)	Dirección dato	Prebuscar	Dir. prebúsqueda	Confianza prebúsqueda	Resultado acceso
0	m2[0][0]	0.0	No	-	0	Fallo
1	m2[1][0]	2.0	No	-	0	Fallo
2	m2[2][0]	4.0	Sí	6.0	1	Fallo
3	m2[3][0]	6.0	Sí	8.0	2 (máx)	Acierto / Fallo parcial
4	m2[4][0]	8.0	Sí	10.0	2 (máx)	Acierto / Fallo parcial
5	m2[5][0]	10.0	Sí	12.0	2 (máx)	Acierto / Fallo parcial
6	m2[6][0]	12.0	Sí	14.0	2 (máx)	Acierto / Fallo parcial
7	m2[7][0]	14.0	Sí	16.0	2 (máx)	Acierto / Fallo parcial
0	m2[0][1]	0.1	Sí	2.1	1	Acierto
1	m2[1][1]	2.1	Sí	4.1	2 (máx)	Acierto

Tabla 5: Tabla ejemplo de funcionamiento del prebuscador stride

En el cuadro 5, se muestra el comportamiento del prebuscador stride en la instrucción 19 del código 4 del bucle interno, durante su primera ejecución, y parte de la segunda. En la iteración 0, no existe una entrada en la *tabla stride* correspondiente a la instrucción, por lo que se añade y no se prebusca nada. No es posible calcular el *stride* puesto que no hay otro acceso con el que comparar, siemplemente se guarda el TAG de la instrucción y la dirección de acceso. En la iteración 1, se calcula el *stride* a partir de la dirección anterior y la de acceso actual, pero no se prebusca nada por falta de confianza en el nuevo patrón. En la iteración 2, se predice que el acceso actual a partir de del último acceso y del *stride* almacenado. En este caso coincide con la dirección de acceso, por lo que se ha confirmado el patrón, aumentando la confianza y se decide prebuscar el siguiente elemento (en este caso 6.0). Durante el resto de iteraciones del bucle, conforme se producen los accesos a los elementos de la matriz, se van prebuscando los bloques siguientes.

En las dos últimas filas de la tabla, se añaden las dos primeras iteraciones de la siguiente ejecución del bucle interno para demostrar la utilidad de realizar una prebúsqueda en caso de fallo si la confianza es alta. En la primera iteración de la segunda ejecución del bucle, se produce un fallo en la predicción, puesto que se recorre la siguiente columna de la matriz. Como la confianza en el patrón que había era mayor a 1, lo que quiere decir una confianza alta, a pesar de fallar la predicción y reducir la confianza, se prebusca el siguiente elemento a partir de la dirección de acceso actual y el *stride* almacenado, y como puede verse acertando en la predicción siguiente. Si se volviese a fallar la predicción, la confianza llegaría a 0 y se desecharía el patrón, repitiendo el proceso de buscar uno nuevo. Con este diseño se permite prebuscar en todos los accesos a memoria de la instrucción como hemos visto, reduciendo los fallos de predicción únicamente a las dos primeras iteraciones de la primera ejecución del bucle y solamente la primera de cada nueva ejecución.

Por otro lado, destacar que al prebuscar el siguiente elemento del patrón, pueden darse dos escenarios: que se produzca un acierto, o que se produzca un fallo parcial (ver Anexo A), esto se debe a que en función del número de ciclos que transcurran entre los dos accesos. No obstante, aunque se produzca un fallo parcial, todos los ciclos de antelación desde que se prebuscó el bloque suponen una optimización en la latencia de acceso.

Por último, indicar que en este ejemplo concreto, la prebúsqueda en la segunda ejecución del bucle no aporta ninguna ventaja respecto a un sistema sin ella debido a que los elementos accedidos son los de la siguiente columna y están en los mismos bloques que los accedidos anteriormente, ya presentes en cache. En otro código, sí podría resultar útil, por ejemplo en el caso de que cada dato ocupase un bloque entero.

C.5. Ejemplo de funcionamiento del prebuscador adaptativo

En este apartado, se aportará un ejemplo ilustrativo del funcionamiento del prebuscador adaptativo. Es preciso recordar que esté prebuscador es una evolución del prebuscador stride, por lo que es conveniente entenderlo primero. Además el código de ejemplo utilizado es el mismo que en el prebuscador stride C.4, donde ya ha sido explicado y se han indicado las consideraciones tomadas. Se ha escogido el mismo ejemplo de código para que su comportamiento pueda ser comparado. Por último, añadir que el número de bloques que se prebuscan coincide con el nivel de confianza en el patrón y la ventaja máxima de prebúsqueda son 5 elementos.

Iteración	Dato accedido (k, j)	Dirección dato	Prebuscar	Dir. prebúsqueda	Confianza prebúsqueda	Resultado acceso
0	m2[0][0]	0.0	No	-	0	Fallo
1	m2[1][0]	2.0	No	-	0	Fallo
2	m2[2][0]	4.0	Sí	{6.0}	1	Fallo
3	m2[3][0]	6.0	Sí	{8.0. 10.0}	2	Acierto / Fallo parcial
4	m2[4][0]	8.0	Sí	{12.0, 14.0, 16.0, 18.0}	4	Acierto / Fallo parcial
5	m2[5][0]	10.0	Sí	{20.0} Máx. distancia	8 (máx)	Acierto / Fallo parcial
6	m2[6{][0]	12.0	Sí	{22.0} Máx. distancia	8 (máx)	Acierto / Fallo parcial
7	m2[7][0]	14.0	Sí	{24.0} Máx. distancia	8 (máx)	Acierto / Fallo parcial
0	m2[0][1]	0.1	Sí	{2.1}	1	Acierto
1	m2[1][1]	2.1	Sí	{4.1. 6.1}	2	Acierto

Tabla 6: Tabla ejemplo de funcionamiento del prebuscador adaptativo

En el cuadro 6 se muestra el funcionamiento del predictor, donde únicamente se van a explicar las novedades respecto al predictor stride. El primer punto que destaca es que se prebusca más bloques cuanta mayor es la confianza existente en el patrón. Esta agresividad es la principal característica de este prebuscador, además, no se buscan los bloques a partir de la dirección de acceso actual, si no que se realiza a partir de prebúsquedas anteriores, con la finalidad de no repetir peticiones. Por otro lado, la distancia máxima a la que se puede prebuscar (5 elementos) se alcanza en la iteración 4, por lo que a partir de ese memento solo se prebusca el siguiente elemento en el patrón. El objetivo de esta limitación es reducir los bloques innecesarios que se solicitan a la cache con este prebuscador. Por último, un fallo en la predicción reduce la confianza en el patrón a la mínima sin desecharlo para ser capaz de reaccionar rápidamente a cambios en los patrones de acceso.

En este ejemplo concreto, el prebuscador adaptativo es demasiado agresivo y trae muchos bloques que no se necesitan. No obstante para patrones más largos esta agresividad supone un gran aumento en el rendimiento.

D. Aplicaciones SPEC

Los *benchmarks* ejecutados[6] son:

SPECrate 2017 Integer (10 *benchmarks*)

1. **500.perlbench_r**: Intérprete del lenguaje Pearl.
2. **502.gcc_r**: Compilador GNU C.
3. **505.mcf_r**: Planeador de rutas.
4. **520.omnetpp_r**: Simulador de eventos discretos.
5. **523.xalancbmk_r**: Conversión XML a HTML mediante XSLT.
6. **525.x264_r**: Compresión de vídeo.
7. **531.deepsjeng_r**: IA: Búsqueda en árbol alfa-beta (Ajedrez).
8. **541.leela_r**: IA: Búsqueda en árbol Monte Carlo (Go)
9. **548.exchange2_r**: IA: Generador de soluciones recursivo (Sudoku).
10. **557.xz_r**: Compresión de datos.

SPECrate 2017 Floating Point(13 *benchmarks*)

1. **503.bwaves_r**: Simulación de explosiones.
2. **507.cactuBSSN_r**: Física: relatividad.
3. **508.namd_r**: Dinámica molecular.
4. **510.parest_r**: Impresión biomédica: tomografía óptica con elementos finitos.
5. **511.povray_r**: Ray tracing.
6. **519.lbm_r**: Dinámica de fluidos.
7. **521.wrf_r**: Predicción del tiempo.
8. **526.blender_r**: Representación y animación 3D.
9. **527.cam4_r**: Simulación atmosférica.
10. **538.imagick_r**: Manipulación de imagen.
11. **544.nab_r**: Dinámica molecular.
12. **549.fotonik3d_r**: Electromagnética computacional.
13. **554.roms_r**: Simulación oceánica.

E. Características ortogonales de la prebúsqueda de datos

Existen una serie de características independientes del propio diseño del prebuscador que afectan a la prebúsqueda . Tienen un impacto considerable, ya sea en términos de eficacia en la prebúsqueda como en el uso de la memoria.

E.1. Políticas de desencadenamiento de prebúsqueda

Probablemente se trate de la característica más importante, decidir cuando hay que prebuscar. Existen tres posibilidades[1], [2]:

- **Always**: Consiste en prebuscar siempre que haya un acceso a memoria, aumentando mucho su uso. De esta forma se pide el siguiente bloque tantas veces como se demande el anterior, sobrecargando innecesariamente de peticiones a la memoria cache.
- **On miss**: Solo se prebusca cuando hay un fallo en memoria al pedir el dato. Esto es en el primer acceso a un bloque. Tiene el inconveniente de que solo se consiguen prebuscar la mitad de los bloques nuevos. Cuando se falla, se pide el siguiente bloque de patrón, pero el siguiente a ese fallará.
- **Tagged**: Se desencadena prebúsqueda bajo dos condiciones: fallo e cache por demanda y acierto en un bloque prebuscado al que se accede por primera vez. Permiten prebuscar todos los bloques sin hacer más peticiones de las imprescindibles, siendo el más eficiente. El gran inconveniente que tiene es que se necesita modificar la memoria cache para añadir un bit extra por contenedor que identifica si se accede o no por primera vez.

E.2. Granularidad del prebuscador

Indica que direcciones utiliza el prebuscador, estas pueden ser la dirección del dato o la dirección del bloque. Normalmente, en caches L1 es posible utilizar ambas. En niveles superiores suele disponerse únicamente de la dirección del bloque. Si se busca aprender patrones es más sencillo utilizar la dirección. Por ejemplo, si se usa la dirección del bloque con una política de prebúsqueda *always*, resulta muy complicado encontrar el patrón, mientras que con política *tagged*, resulta indiferente el uso de una dirección otra.

E.3. Políticas de inserción en tabla

Es una política exclusiva de los prebuscadores que utilizan una tabla indexada por PC. Consiste en decidir que hacer cuando una instrucción no tiene su información asociada en la tabla, es decir el *TAG* no coincide y/o la entrada no tiene validez. La solución convencional es reemplazar la entrada con la información de la nueva instrucción, pero existe otra alternativa.

La otra posibilidad consiste en insertar únicamente aquellas instrucciones que no tengan entrada en la tabla si y solo si han provocado un fallo en memoria cache [33]. Dicho con otras palabras, las instrucciones que no tienen entrada correspondiente en la tabla y producen aciertos en cache nunca se insertan. Esto es útil porque algunas de las instrucciones que acceden a memoria están correlacionadas con otras. Un ejemplo sencillo de correlación es imaginar la siguiente situación: dentro de un bucle se encuentran la instrucción A y B, las cuáles recorren la misma estructura, A accede a los elementos pares y B a los impares(el tamaño de los elementos es menor que el del

bloque). Es lógico pensar que únicamente utilizando las direcciones solicitadas por A se consigue prebuscar los accesos de B, convirtiéndolos en aciertos. Este es un sencillo ejemplo pero la idea de que hay instrucciones cuyos accesos están correlacionados con los de otras es genérico.

Esta solución presenta como ventaja la reducción en el número de entradas necesarias en las tablas de los prebuscadores stride y adaptativo, las cuales suponen una parte sustancial del coste de su implementación.

F. Ejemplo de representación gráfica y textual en VHDL de un *Full-adder*

Un *Full-adder (FA)* o sumador completo es un circuito digital encargado de sumar dos bits (A, B) junto al acarreo heredado (Cin) y propagar el resultado de la suma (S) junto al acarreo generado (Cout). En la Figura 24 se muestra su representación gráfica del FA utilizando puertas lógicas que definen el valor de las señales de salida en función de sus señales de entrada. La Figura 25 muestra una representación textual o especificación del mismo FA en VHDL.

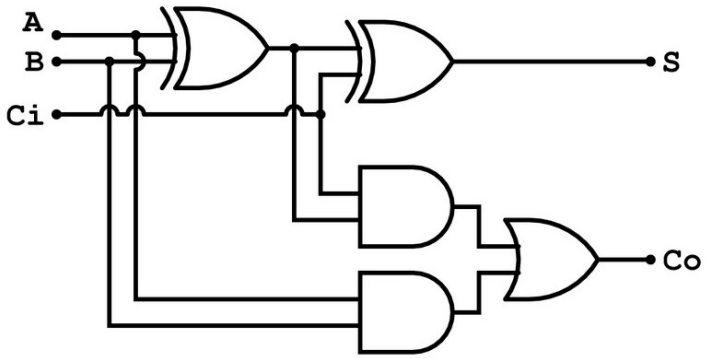

Figura 24: Representación gráfica de un *Full-adder* [34]

```
1  entity full_adder is
2   Port (
3   A : in STD_LOGIC;
4   B : in STD_LOGIC;
5   Cin : in STD_LOGIC;
6   S : out STD_LOGIC;
7   Cout : out STD_LOGIC
8   );
9  end full_adder;
10
11 architecture gate_level of full_adder is
12
13 begin
14
15  S <= A XOR B XOR Cin;
16  Cout <= (A AND B) OR (Cin AND A) OR (Cin AND B);
17
18 end gate_level;
```

Figura 25: *Full-adder* de en VHDL. Implementación de *allaboutfpga* [35]

G. Ejemplo de uso de genéricos en VHDL

Este pequeño ejemplo muestra la utilidad del uso de parámetros genéricos en un diseño VHDL. En la Figura 26 se implementa un registro con señal de *reset* asíncrona. En este diseño el parámetro genérico es la anchura del registro, que si no se especifica un valor para dicho parámetro es 64. La Figura 27 muestra como se instancia un componente con un parámetro genérico. En este caso *reg1* es un registro de 64 bits y *reg2* de 32 bits. De esta manera se puede modificar muy fácilmente el tamaño de un componente en un diseño o reutilizarlo en distintos tamaños, como es el caso del ejemplo.

```
1  entity reg is
2      GENERIC(d_width :   INTEGER := 64);     -- anchura del registro
3      Port ( Din : in  STD_LOGIC_VECTOR(d_width-1 DOWNTO 0);
4             clk : in  STD_LOGIC;
5             reset : in  STD_LOGIC;
6             load : in  STD_LOGIC;
7             Dout : out  STD_LOGIC_VECTOR(d_width-1 DOWNTO 0));
8  end reg;
9
10  architecture Behavioral of reg is
11 begin
12 SYNC_PROC: process (clk, reset)
13    begin
14      if (reset = '1') then
15         Dout <= (others => '0');
16      elsif rising_edge(clk) then
17         if(load = '1') then
18            Dout <= Din;
19         end if;
20      end if;
21    end process;
22 end Behavioral;
```

Figura 26: Implementación de un registro de anchura genérica

```
25 reg1:   reg generic map (d_width => 64)
26            port map(Din => reg1_out, clk => clk_i, reset => rstn_i, load => load_reg1, Dout => reg1_o);
27
28 reg2:   reg generic map (d_width => 32)
29            port map(Din => reg2_out, clk => clk_i, reset => rstn_i, load => load_reg2, Dout => reg2_o);
```

Figura 27: Dos instanciaciones de un registro genérico

H. Informe de Síntesis en FPGA

```
1
2  Version 2
3  ==========
4      32 reg:
5
6  HDL Synthesis Report
7
8  Macro Statistics
9  # Counters                                          : 1
10  6-bit updown counter                               : 1
11 # Registers                                         : 33
12  64-bit register                                    : 33
13
14 Device utilization summary:
15 ---------------------------
16
17 Selected Device : 5vlx50tff1136-1
18
19
20 Slice Logic Utilization:
21  Number of Slice Registers:            2120  out of  28800     7%
22  Number of Slice LUTs:                 3851  out of  28800    13%
23     Number used as Logic:              3851  out of  28800    13%
24
25 Slice Logic Distribution:
26  Number of LUT Flip Flop pairs used:   3853
27    Number with an unused Flip Flop:    1733  out of   3853    44%
28    Number with an unused LUT:             2  out of   3853     0%
29    Number of fully used LUT-FF pairs:  2118  out of   3853    54%
30    Number of unique control sets:        35
31
32 Timing Summary:
33 ---------------
34 Speed Grade: -1
35
36    Minimum period: 3.207ns (Maximum Frequency: 311.818MHz)
37    Minimum input arrival time before clock: 3.882ns
38    Maximum output required time after clock: 6.041ns
39    Maximum combinational path delay: 6.682ns
40
41 Version 3
42 ==========
43    32 reg:
44
45 Device utilization summary:
46 ---------------------------
47
48 Selected Device : 5vlx50tff1136-1
49
50
51 Slice Logic Utilization:
52  Number of Slice Registers:            2182  out of  28800     7%
53  Number of Slice LUTs:                 2822  out of  28800     9%
54     Number used as Logic:              2822  out of  28800     9%
55
56 Slice Logic Distribution:
57  Number of LUT Flip Flop pairs used:   2822
58    Number with an unused Flip Flop:     640  out of   2822    22%
59    Number with an unused LUT:             0  out of   2822     0%
60    Number of fully used LUT-FF pairs:  2182  out of   2822    77%
```

```
   Number of unique control sets:        28

IO Utilization:
 Number of IOs:                         326
 Number of bonded IOBs:                 326  out of    480    67%

Specific Feature Utilization:
 Number of BUFG/BUFGCTRLs:                1  out of     32     3%

Timing Summary:
---------------
Speed Grade: -1

   Minimum period: 3.556ns (Maximum Frequency: 281.201MHz)
   Minimum input arrival time before clock: 4.418ns
   Maximum output required time after clock: 6.188ns
   Maximum combinational path delay: 7.144ns

=========================================================================
   --
=========================================================================
   4 reg:


--------------------------

Selected Device : 5vlx50tff1136-1


Slice Logic Utilization:
 Number of Slice Registers:             390  out of  28800     1%
 Number of Slice LUTs:                  592  out of  28800     2%
    Number used as Logic:               592  out of  28800     2%

Slice Logic Distribution:
 Number of LUT Flip Flop pairs used:    594
   Number with an unused Flip Flop:     204  out of    594    34%
   Number with an unused LUT:             2  out of    594     0%
   Number of fully used LUT-FF pairs:   388  out of    594    65%
   Number of unique control sets:         8

Timing Summary:
---------------
Speed Grade: -1

   Minimum period: 2.212ns (Maximum Frequency: 452.033MHz)
   Minimum input arrival time before clock: 3.053ns
   Maximum output required time after clock: 5.659ns
   Maximum combinational path delay: 6.022ns

   =========================================================================
```

I. Gráficas de resultados de simulación

I.1. 523.xalancbmk_r

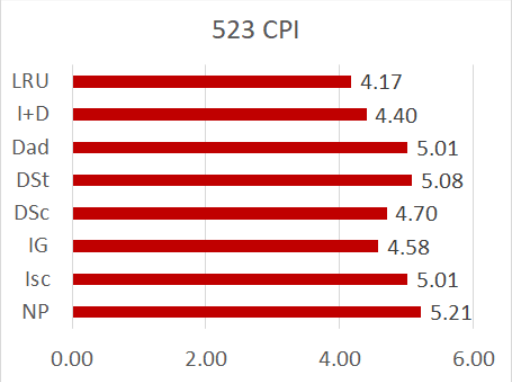

Figura 28

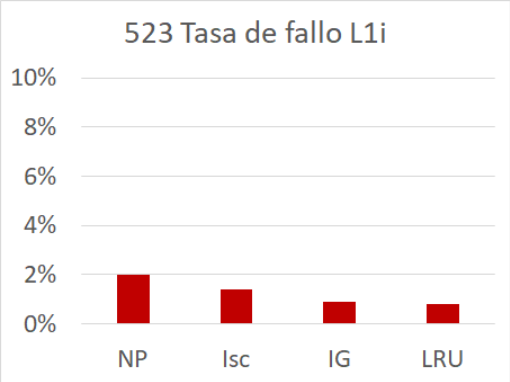

Figura 29

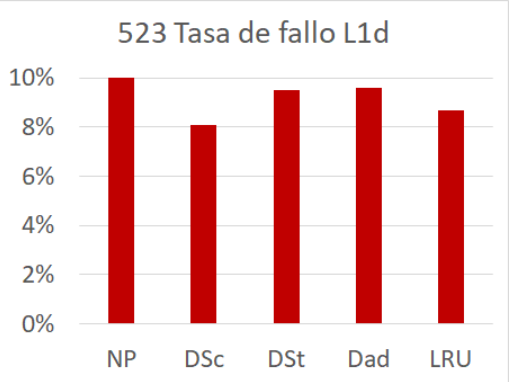

Figura 30

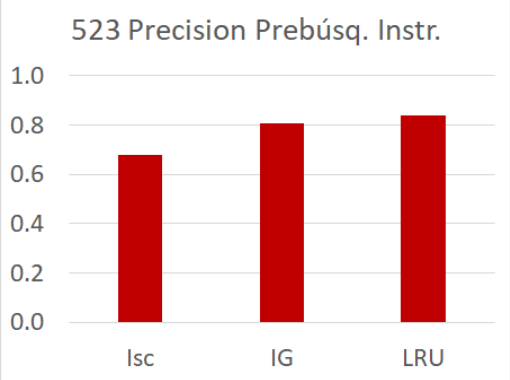

Figura 31

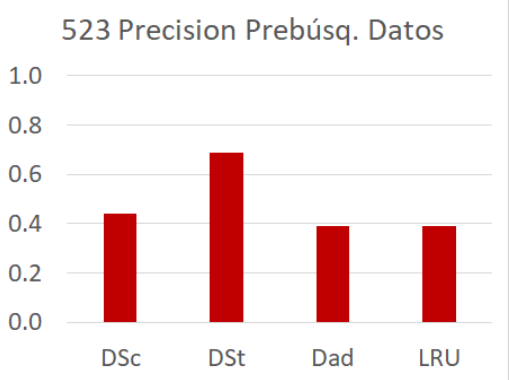

Figura 32

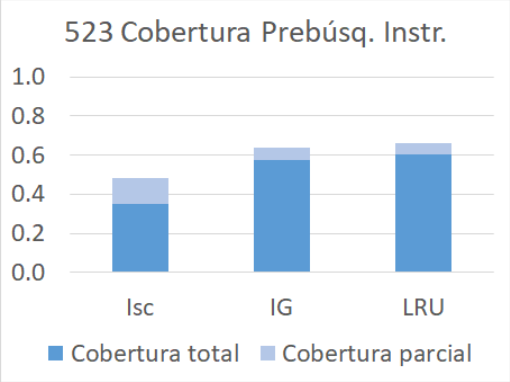

Figura 33

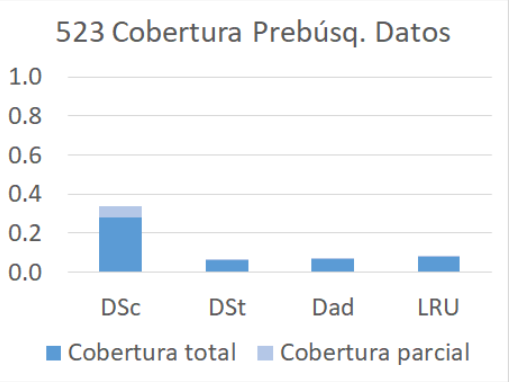

Figura 34

I.2. 531.deepsjeng_r

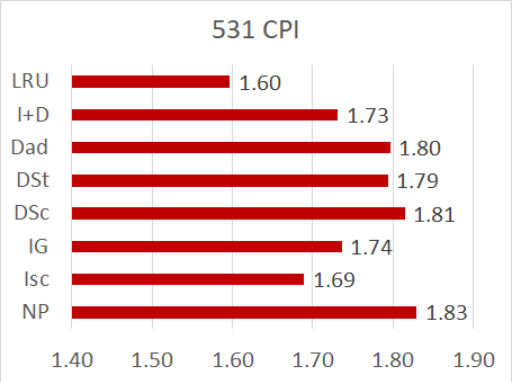

Figura 35

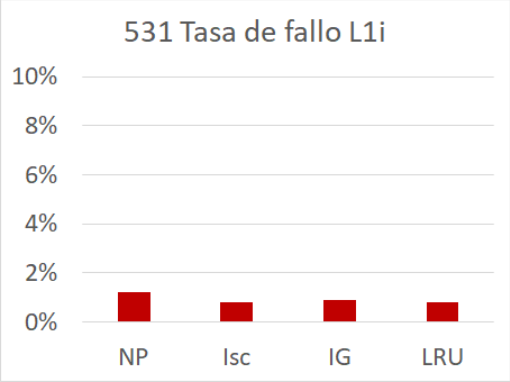

Figura 36

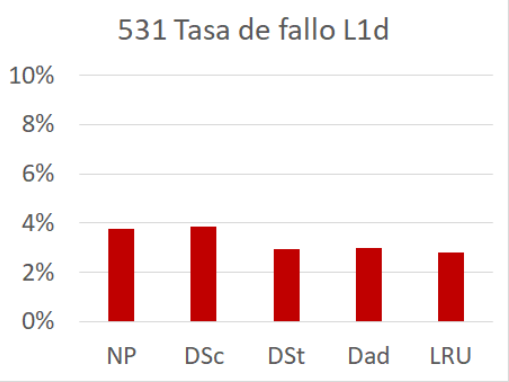

Figura 37

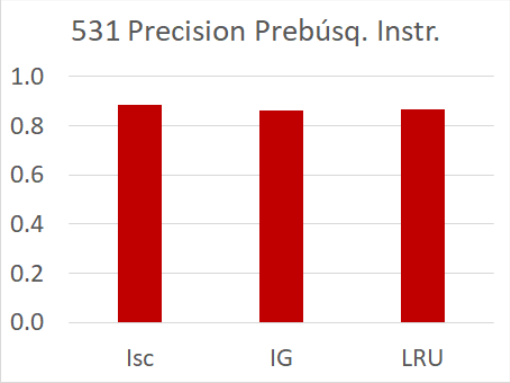

Figura 38

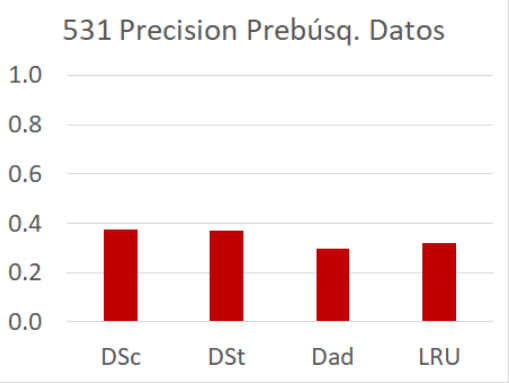

Figura 39

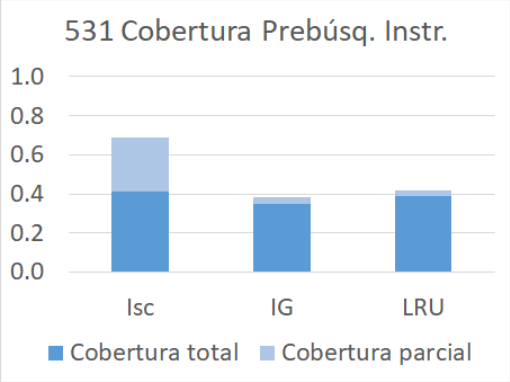

Figura 40

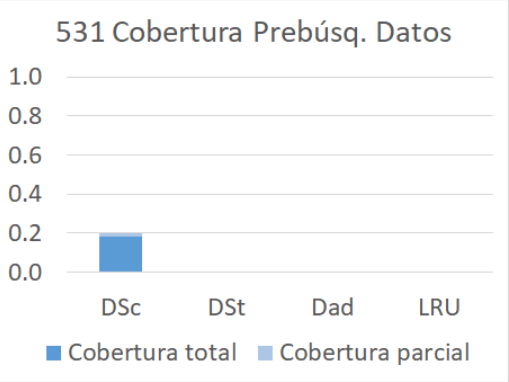

Figura 41

I.3. 541.leela_r

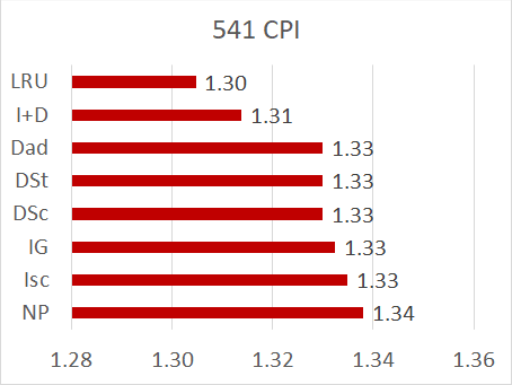

Figura 42

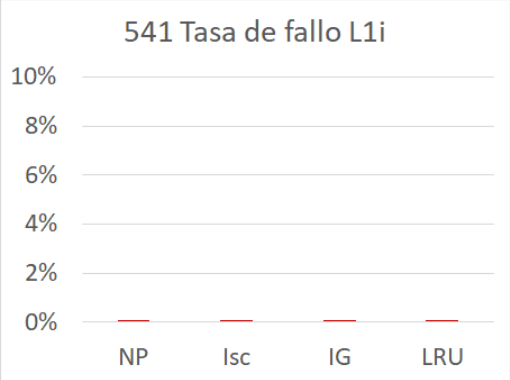

Figura 43

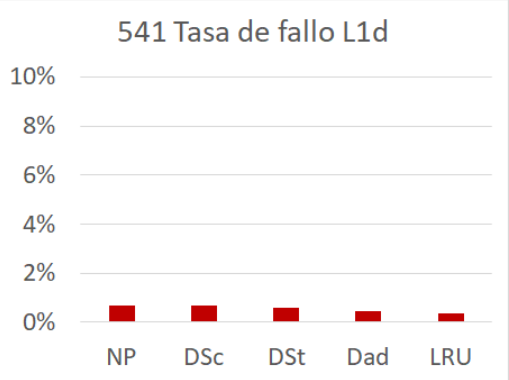

Figura 44

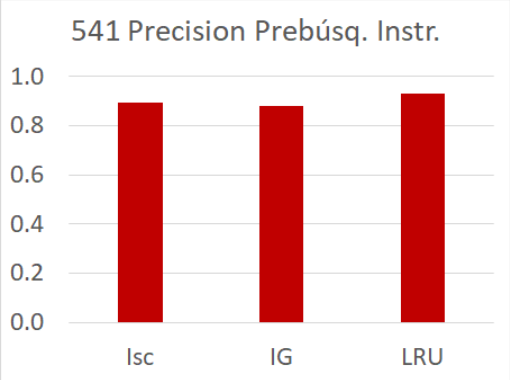

Figura 45

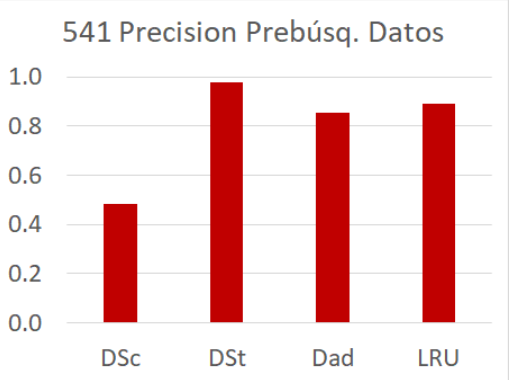

Figura 46

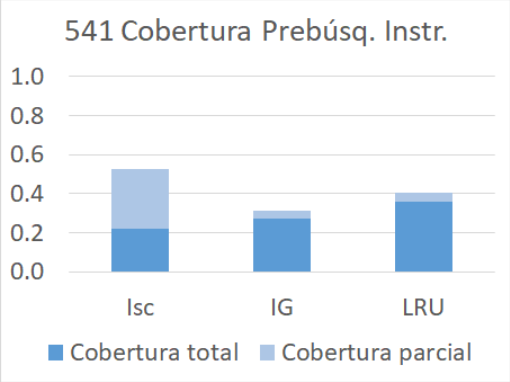

Figura 47

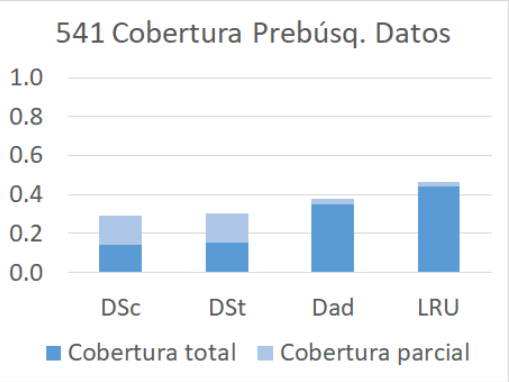

Figura 48

I.4. 549.fotonik3d_r

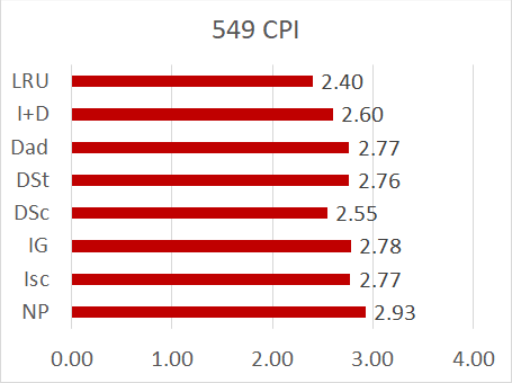

Figura 49

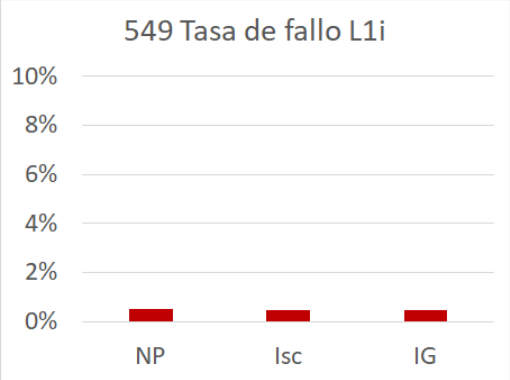

Figura 50

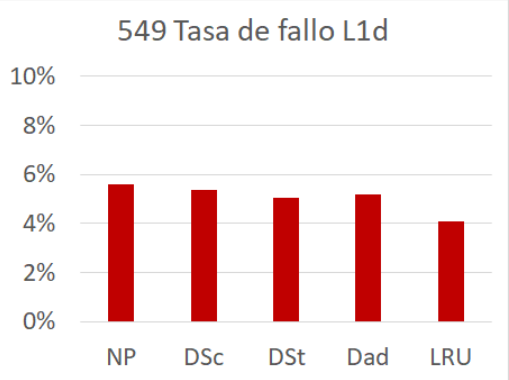

Figura 51

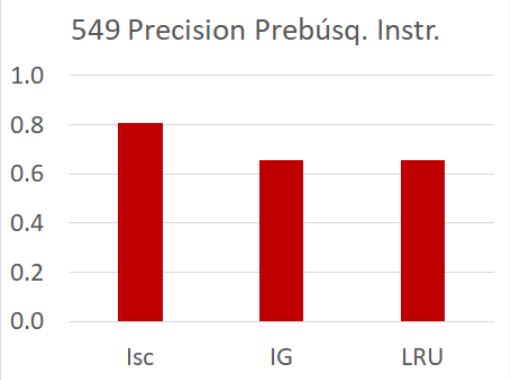

Figura 52

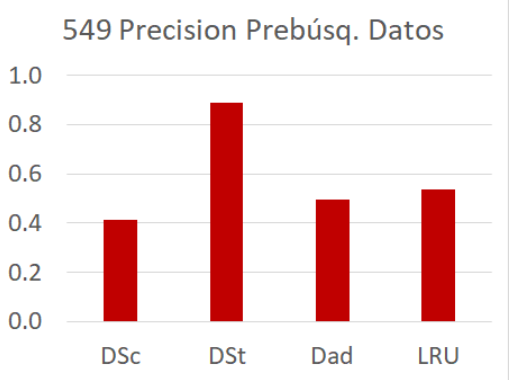

Figura 53

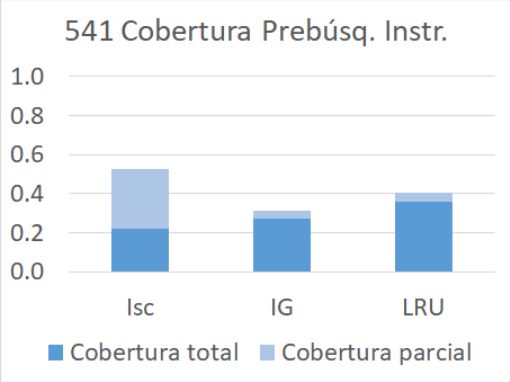

Figura 54

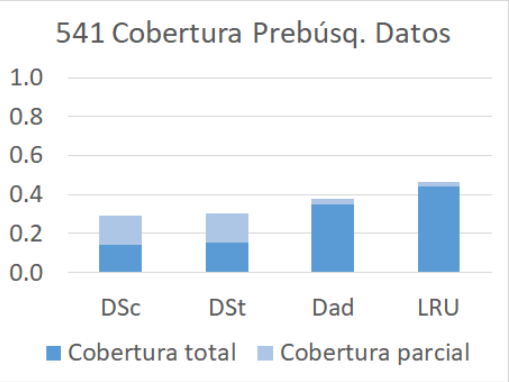

Figura 55

I.5. 557.xz_r

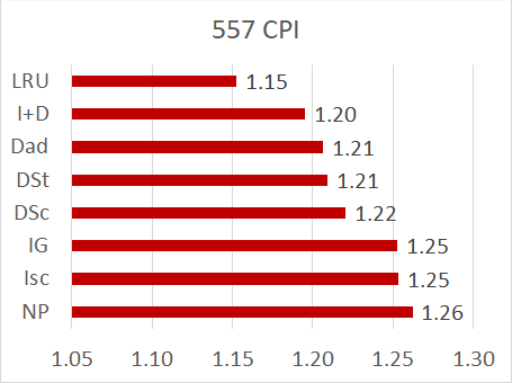

Figura 56

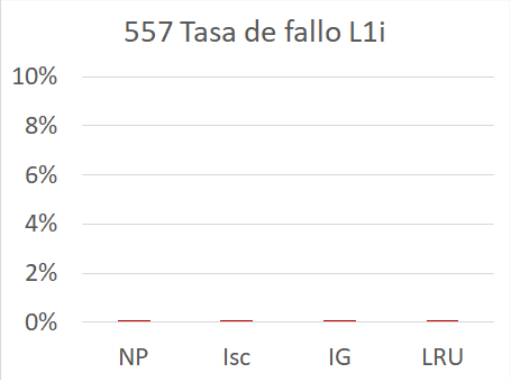

Figura 57

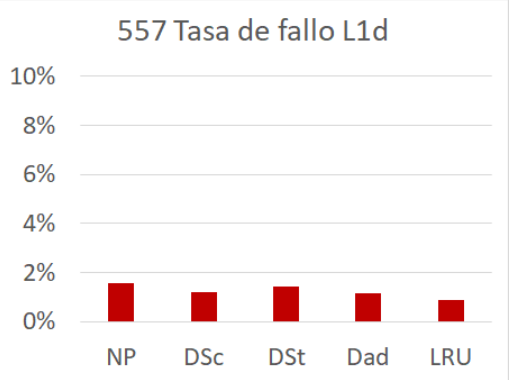

Figura 58

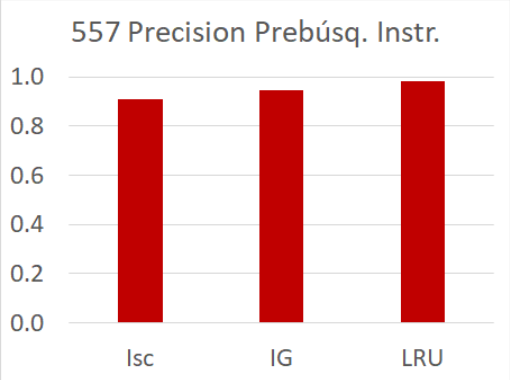

Figura 59

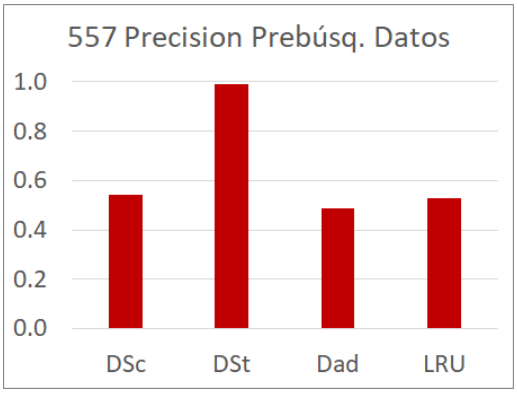

Figura 60

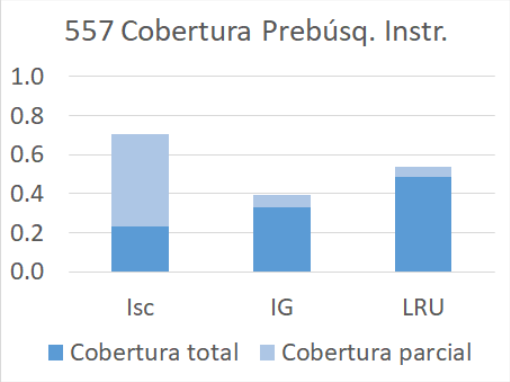

Figura 61

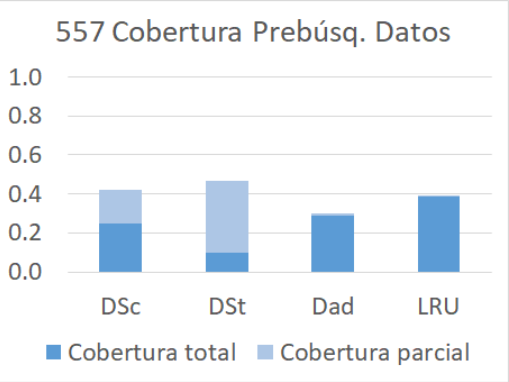

Figura 62

I.6. matrizf

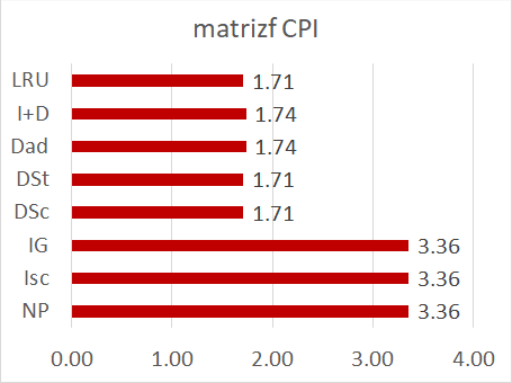

Figura 63

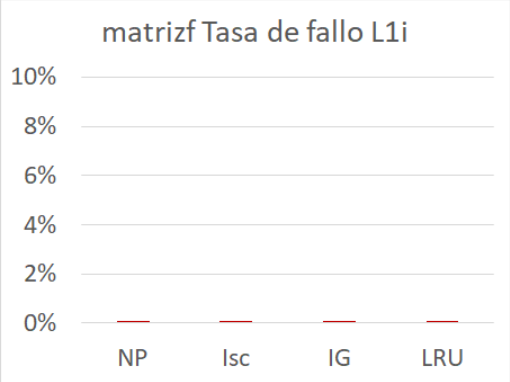

Figura 64

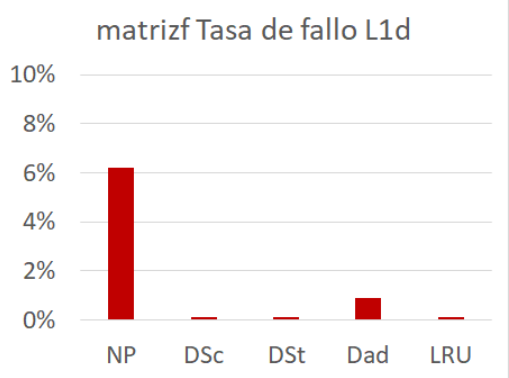

Figura 65

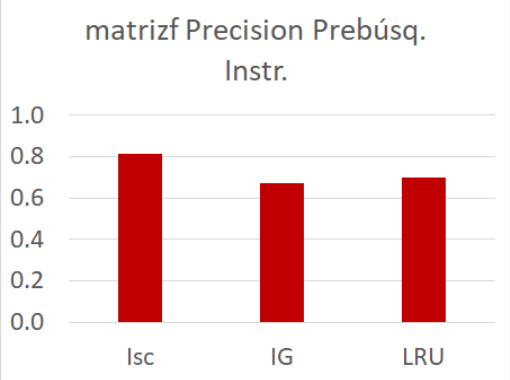

Figura 66

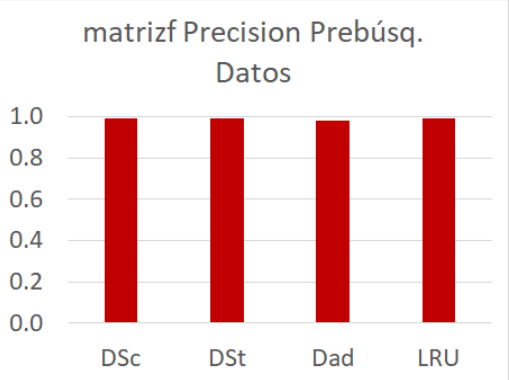

Figura 67

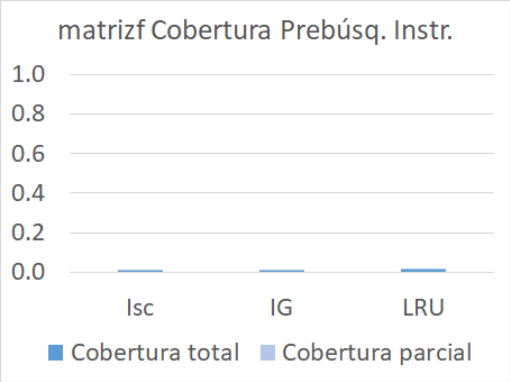

Figura 68

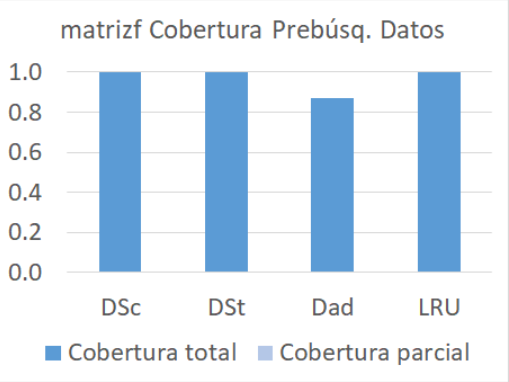

Figura 69

I.7. matrizc

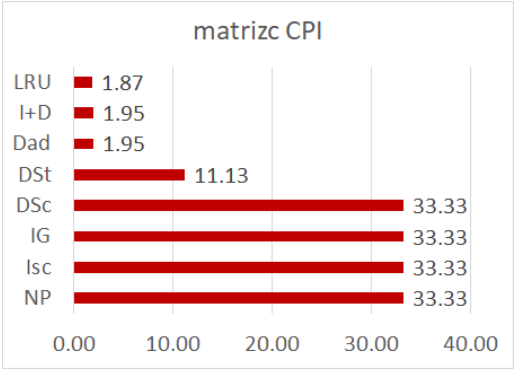

Figura 70

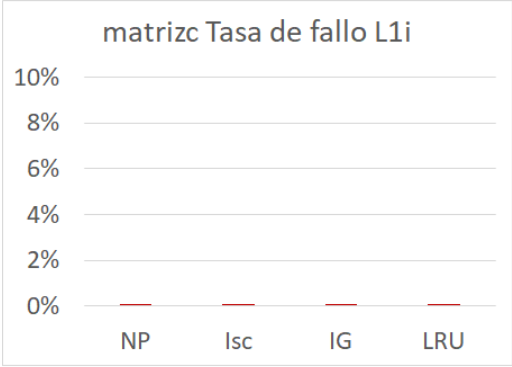

Figura 71

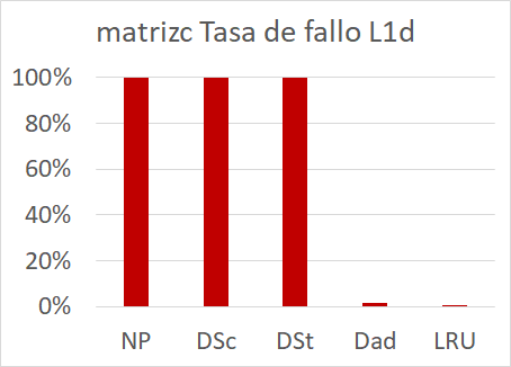

Figura 72

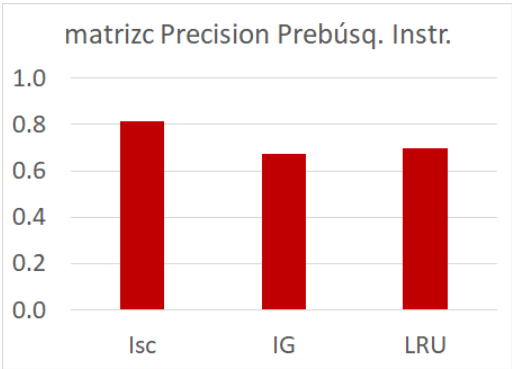

Figura 73

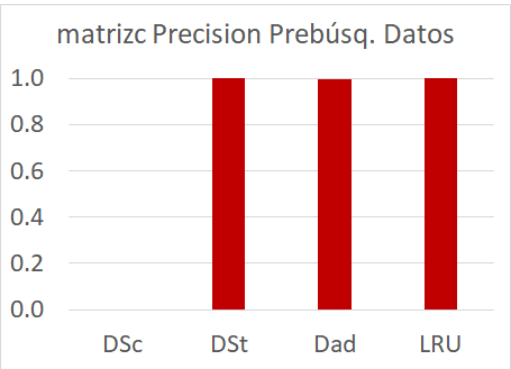

Figura 74

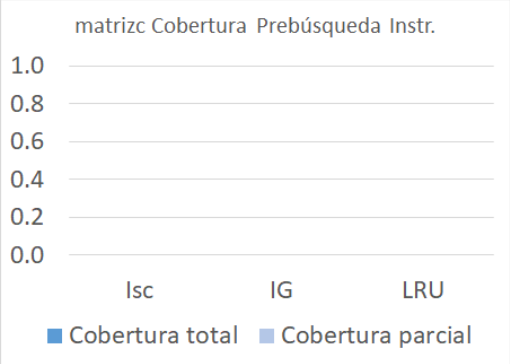

Figura 75

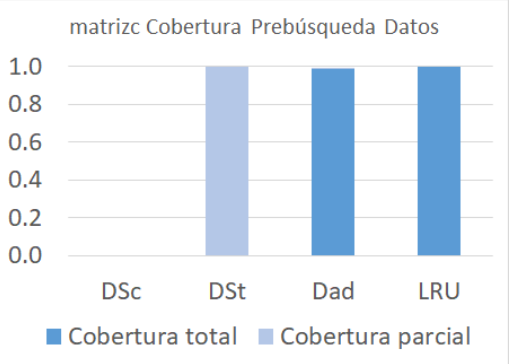

Figura 76