Sistemas Informáticos
Curso 2005-2006

ESTUDIO COMPARADO DE ARQUITECTURAS GALS-SMT (Globalmente Asíncronos Localmente Síncronos – Simultaneous Multithreading)

Omar Aragón
César Boullosa Serrano
Tania Orell Orgaz

Dirigido por:
Juan Lanchares Dávila
Dpto. Arquitectura de Computadores y Automática (DACYA)

Facultad de informática
Universidad complutense de Madrid

RESUMEN:

En este trabajo se realizará un estudio de una arquitectura resultado de la unión de un diseño SMT y otro GALS (MCD). Esta arquitectura nunca ha sido probada, por lo que los parámetros iniciales no están ajustados para aprovechar las ventajas que ofrece un diseño SMT.

Dividiremos el trabajo en dos partes bien diferenciadas. En la primera sintonizaremos los parámetros de la arquitectura para un mejor aprovechamiento del diseño SMT. Una vez conseguido esto, en la segunda parte, estudiaremos los dominios de reloj óptimos en los que tendríamos que dividir el sistema. Para esto, realizaremos un estudio de los canales de comunicación, buscando aquellos que tienen un mayor impacto en el rendimiento para descubrir los cuellos de botella.

PALABRAS CLAVE:

SMT, GALS, MCD, reloj, canales, penalizaciones, dominios.

SUMMARY:

In this work, we will cary out a study of an architecture which is the result of the link of an SMT design and a GALS design (MCD). This architecture has never been tested, therefore, the initial parameters are not adjusted to take advantage of an SMT design.

We will split the work in two clearly different parts. In the first one, the parameters of the architecture will be fit in order to get a better use of the SMT design. Once this target is achieved, we will study the optimal clock domains in which the system should be split. To achieve this goal, we will carry out a study about the communication channels, looking for those ones with higher impact on the performance to discover the bottlenecks.

KEY WORDS:

SMT, GALS, MCD, clock, channels, penalties, domains.

ÍNDICE

1) INTRODUCCIÓN

Una característica importante de los diseños de procesadores que hoy en día dominan el mercado es el uso de un reloj global (son síncronos). La simulación de estos tipos de procesadores es sencilla, y hay muchas herramientas disponibles para su diseño. Sin embargo, el uso de este reloj global, hace que su señal tenga que llegar a todos los módulos del procesador, lo que supone mayor esfuerzo de diseño, a la vez que implica una mayor necesidad de área y un mayor consumo de potencia.

Un pensamiento lógico para evitar todos los problemas derivados del reloj es utilizar un procesador totalmente asíncrono, sin embargo su desarrollo presenta graves problemas al no existir herramientas de diseño aceptadas y eficaces.

Una solución a los problemas generados por los dos tipos de diseño de procesadores mencionados anteriormente, es usar diseños GALS (globalmente asíncronos, localmente síncronos). De esta forma se podrían combinar las ventajas de ambos y minimizar sus inconvenientes. El circuito se divide en varios dominios de reloj, cada uno trabajando a una frecuencia diferente. La comunicación entre los distintos módulos se realiza de forma asíncrona, evitando así el uso de un reloj global. Además, el hecho de que cada dominio pueda trabajar a una frecuencia diferente, hace que se puedan elegir aquellas bajo las cuales los dominios presentan mejores comportamientos.

Por otro lado, para aumentar las capacidades del procesador, hace falta incrementar el paralelismo de todas las formas posibles. Estas formas son el paralelismo a nivel de instrucción, y a nivel de hilo. Una microarquitectura capaz de conseguir esto, es la Simultaneous Multithreading. Este diseño, hereda de los superescalares la habilidad de lanzar varias instrucciones cada ciclo, y de los multihilo, la habilidad de la coexistencia de varios hilos de ejecución en el pipeline. Esto hace que haya una mayor eficiencia en el uso de los recursos, y que sea más sencillo apantallar las paradas de un único hilo.

Vistas las ventajas de ambos planteamientos, parece lógico pensar que el uso de una arquitectura SMT-GALS pueda ser ventajoso en cuanto a la distribución del reloj a la par que incrementa el rendimiento debido a la ejecución de múltiples hilos simultáneamente.Nuestro estudio se centrará en una arquitectura de éste tipo.

El diseño GALS, aunque hace desaparecer la dificultad del uso de un reloj global, no ofrece una mejora en el rendimiento debido a las penalizaciones. Pero esta ligera degradación del rendimiento puede ser compensada por el hecho de que la arquitectura sea también SMT.

En este trabajo, partiendo de una arquitectura SMT-GALS con parámetros no ajustados para un correcto funcionamiento SMT, los sintonizaremos para obtener una arquitectura con buen rendimiento en ejecuciones con varios hilos. Una vez la tenemos, nuestro trabajo se centrará en la búsqueda de los dominios de reloj óptimos, estudiando las penalizaciones que aparecen en los canales de comunicación dado el carácter GALS de la arquitectura, y su impacto en el rendimiento.

Esta memoria se divide en tres partes:

En la primera parte comentamos ampliamente las características del mercado actualmente en lo que se refiere a GALS y a Simultaneous Multithreading. En la segunda parte explicamos los detalles de la microarquitectura que hemos usado. La tercera parte se centra en nuestros resultados, explicando el método de simulación usado, así como los resultados obtenidos y sus conclusiones

2) ESTADO DEL ARTE

2.1) Los diseños GALS (Globalmente Asíncronos Localmente Síncronos)

En el diseño de procesadores nos encontramos con dos alternativas extremas, los procesadores totalmente síncronos y los totalmente asíncronos, que a continuación pasamos a describir.

2.1.1) Procesadores totalmente síncronos:

Son diseños en los que un único reloj se distribuye a lo largo de todo el sistema. La progresión del desarrollo de los circuitos síncronos se basa en la discretización del tiempo. Esto facilita la descripción de los circuitos, ya que el diseñador hace la hipótesis de que todas las operaciones del circuito acaban en un determinado tiempo para ser muestreadas con el siguiente pulso de reloj.

Además, la observación de los valores en los ciclos de reloj, facilita la simulación y el desarrollo del diseño. La verificación del mismo se convierte en un problema de estudiar los retardos en las funciones combinacionales lógicas entre registros, y este proceso puede ser automatizado. El estilo de diseño síncrono en conjunción con los lenguajes de descripción de hardware de alto nivel, las herramientas elaboradas y los avances tecnológicos concernientes a la densidad de integración, han permitido grandes avances para el diseño y el desarrollo de los computadores. Por todo esto los diseños síncronos siguen dominando el mercado en la actualidad.

En contrapartida, el uso de un reloj global, que tiene que alimentar a todos y cada uno de los módulos que componen el diseño total, requiere un cableado del reloj más complejo para que éste alcance a todos los módulos. Ello supone un gran esfuerzo de diseño y una necesidad de ocupar mayor área, desencadenando una serie de inconvenientes y problemas importantes que describiremos a continuación.

El retardo del reloj:

Debido a que un único reloj tiene que llegar a todos los módulos incluidos los más lejanos (el tamaño de la oblea es cada vez mayor), se produce un problema, el retardo del reloj (tamaños cada vez más grandes, cables cada vez más largos y por consiguiente hay un retardo mayor hasta que la señal del reloj alcanza a todos los módulos), que es una causa importante de la pérdida de rendimiento en un procesador.

La disipación de potencia:

El consumo de potencia es un gran problema que afecta negativamente al rendimiento del procesador porque produce el calentamiento de ciertas partes del chip lo que tiene como efecto que éste funcione más lentamente. Esta disipación se debe a dos motivos:

- Cada señal del reloj que llega a un módulo provoca que los transistores del módulo cambien de estado (aunque puedan luego volver al estado

anterior), y con cada cambio de estado en un transistor se consume potencia.

- La mayor longitud del cableado implica mayor consumo de potencia en el mismo.

A su vez, el consumo de potencia se ve aumentado debido a que los avances en el diseño de arquitecturas implican un mayor número de transistores, lo que consume más potencia, y al mismo tiempo se descubren mejores tecnologías que permiten un mayor número de transistores por unidad de superficie, lo que hace que se incremente el consumo por unidad de área.

Una de las soluciones planteadas para evitar este consumo de potencia es inhabilitar el reloj en aquellas zonas que se encuentran inoperativas, pero los resultados no fueron suficientes. Todo eso hizo que los investigadores vieran en los diseños totalmente asíncronos una atractiva alternativa de diseño.

Hoy en día unos de los temas de investigación más importantes en el mundo del diseño de procesadores es cómo reducir este consumo de potencia [1].

El desfase del reloj:

Aunque la distribución del reloj global se intenta diseñar de manera que llegue casi instantáneamente a todos los módulos, este objetivo no siempre se consigue, dando lugar a otro de los problemas que degradan el rendimiento: el desfase del reloj. Éste consiste en que la señal del reloj llega a los distintos módulos en instantes de tiempo diferentes, lo que provoca que un módulo que recibe dicha señal más tarde que el resto cargue información errónea.

Para remediar este problema se invierte un gran esfuerzo de diseño en distribuir el reloj de forma óptima [2] y aún así no siempre es posible eliminar este desfase del reloj.

2.1.2) Procesadores totalmente asíncronos:

Son diseños que no tienen reloj global, eliminando así los problemas derivados del mismo. Pero estos sistemas presentan un grave problema: las herramientas de diseño automático no están todavía muy desarrolladas, por lo que resultan mucho más costosos tanto el diseño como la validación de los sistemas.

Por este motivo, la mayoría de los procesadores totalmente asíncronos existentes son de desarrollo académico, y prácticamente ninguno ha sido comercializado.

Ejemplos:

1. **Caltech** [3]: se diseñó en el año 1988 y fue el primer microprocesador totalmente asíncrono. Es un RISC de 16 bits. No trató ser un diseño de procesador formal, estaba pensado para la experimentación en diseño de circuitos y desarrollo de métodos de prueba. Los diseñadores estimaron la velocidad de ejecución en 15 MIPS.

2. **FAM (**Fully Asynchronous Microprocessor**)** [3]: cuenta con 18 instrucciones en su repertorio. Es experimental, igual que el anterior. Tiene 32 bits en la ruta de datos y 32 registros de 32 bits. Tiene una arquitectura RISC segmentada de cuatro etapas que son: búsqueda, memoria, decodificación y ejecución. La caché contiene instrucciones y datos, y se accede a ella en la primera y cuarta etapa, lo cual produce un conflicto que se resuelve mediante el protocolo de arbitraje. Su velocidad de ejecución son 300 MIPS.

3. **NSR** (Nonsynchronous RISC) [3, 4]: Contiene 5 bloques concurrentes que son análogos a las etapas del pipeline estándar de un sistema síncrono: búsqueda, decodificación, ejecución, acceso a memoria y escritura en registro. Tiene también una cola FIFO para minimizar las paradas causadas por las instrucciones con mayor latencia. Su velocidad de ejecución es de 1'3 MIPS. Su repertorio de instrucciones consta de 16 instrucciones.

4. **CFPP** (Counterflow pipeline processor) [3]: fue desarrollado en 1994 por Sun Microsystems. Las instrucciones fluyen a través del pipeline en una dirección, pero los datos que genera fluyen en la dirección opuesta. El único requisito es que la instrucción primero tiene que encontrarse con los operandos fuente mientras fluyen en sentidos opuestos. Una vez finalizada la instrucción continúa fluyendo hasta llegar al banco de registros, donde se deposita el resultado. Además se inserta el resultado en el flujo opuesto del pipeline.

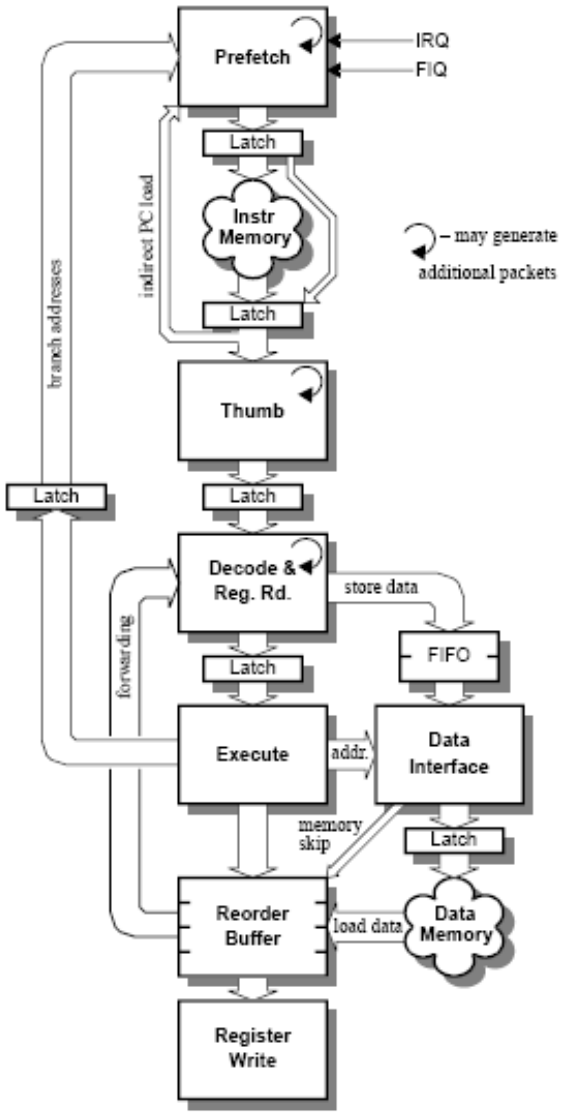
   Ventajas:
   - Ejecución especulativa
   - Ejecución fuera de orden.
   - Asíncronos vs. síncronos: El CFPP fue diseñado para un procesador asíncrono. Sin embargo, puede ser implementado también como un procesador síncrono.
   - Control local: requiere sólo información local para decidir cuando puede avanzar un elemento en el pipeline.
   - Regularidad: tiene una estructura muy regular.
   - Comunicación local: se comunica solamente con los vecinos
   - Modularidad
   - Simplicidad

5. **Amulet:** Amulet3 es un microprocesador con funcionalidad completa que soporta interrupciones y fallos de memoria. Amulet3 se basa en el diseño de una implementación asíncrona del conjunto de instrucciones ARM (*Advanced Risc Machine*). El Amulet1 que se desarrolló durante

1991-1993, mostró que el diseño complejo asíncrono es posible y el Amulet2 que las ventajas de los diseños asíncronos pueden ser realizadas en la práctica.

Amulet3 (1996-1998) se desarrolló para establecer la viabilidad comercial del diseño asíncrono. El objetivo del proyecto Amulet3 es producir una implementación asíncrona de una arquitectura ARM competitiva en términos de eficiencia en cuanto a potencia y ejecución con el último núcleo ARM basado en reloj. Es de los pocos procesadores comerciales totalmente asíncronos.

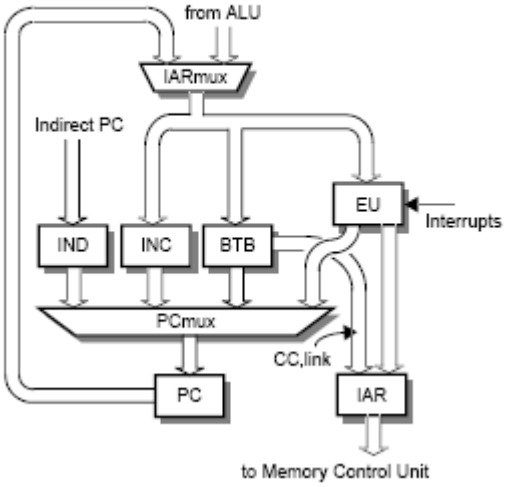
La ruta de datos del Amulet3 es:



- *La unidad de prefetch*:

  Es la responsable de generar direcciones para la memoria de instrucciones que se envían mediante el registro de instrucción de dirección (IAR, en la siguiente figura). La unidad de prefetch tiene una organización paralela, y computa especulativamente todas las salidas en

paralelo para después seleccionar el curso apropiado de la acción en el multiplexor final. Aunque esta especulación causa actividad innecesaria (y por lo tanto consume más potencia) es necesario para tener el rendimiento deseado.



Normalmente las direcciones de salida de una secuencia ascendente son un simple bucle que contiene un incrementador. Cuando hay un salto, este bucle puede ser interrumpido asíncronamente y cargado con una nueva dirección de la ALU.

Características de la unidad de prefetch:

- Predicción de saltos:
  Usa el "branch target buffer" (BTB), que predice que el salto siempre se toma.
- Paradas e interrupciones:
  Cuando llega una interrupción a la etapa prefetch, el contador de programa se carga con la dirección de la rutina de servicio (que se genera en la unidad EU de la figura 2) y comienza la etapa de prefetch para este nuevo código.
- Saltos indirectos:
  La unidad de ejecución pasa el valor de la dirección del load del PC a la unidad de prefetch mediante el camino de la dirección de salto en paralelo con la iniciación de otras transferencias de registros en el interfaz de datos. Este valor vuelve a la unidad de prefetch mediante IND de la figura 2, donde comienza la etapa de prefetch.

- *Decodificador Thumb:*
  Las instrucciones ARM, de 32 bits, más usadas se pueden comprimir a un formato Thumb, de 16 bits. Éstas pueden a su vez descomprimirse. Este decodificador recibe paquetes de instrucciones de 32 bits que pueden contener una única instrucción de este tamaño, dos

Thumb de 16 bits o una excepción. Las que recibe en formato Thumb, las descomprime antes de enviarlas a la siguiente etapa.

- *Decodificador de instrucciones:*

  Este decodificador es la parte más compleja del sistema. La etapa latch entre la unidad de decodificación y la de ejecución (uno de los caminos más largos en el procesador) comprende varios elementos dispares que pueden operar en distintos tiempos. Esto ayuda a reducir las penalizaciones debidas a forwarding, y es también un intento de minimizar el efecto del ruido electromagnético emitido en esta etapa.

- *Unidad de ejecución:*

  Esta unidad elige primero qué instrucción será completada. Una vez tiene los operados, puede ejecutar la operación. La unidad de ejecución puede enviar también una dirección a la memoria de datos.

  Amulet3 lanza las instrucciones en orden, pero luego se dividen en dos. Las operaciones internas se retiran inmediatamente a un buffer de reordenamiento donde sus resultados están disponibles para forwarding. Las instrucciones load y store se envían a la interfaz de datos, donde se completa el acceso a memoria.

- *Interfaz de datos:*

  La interfaz de datos es la responsable de enviar transferencias de memoria, y opera semi-autónomamente. Como se encuentra en un pipeline separado de la unidad de ejecución, recibe sus propias instrucciones sin tener en cuenta si se van a ejecutar o no.

- *Buffer de reordenamiento:*

  Implementa una forma de renombramiento de registros. Los paquetes de datos llevan su propia posición de destino dentro del buffer y son de esta forma clasificados ordenadamente. El buffer de reordenamiento se vacía, en orden, al banco de registros asegurando que cualquier cambio de estado permanente, ocurra en el orden de lanzamiento de las instrucciones.

- *Registro de writeback:*

  Si las entradas del buffer de reordenamiento están marcadas como inválidas, las descarta, en otro caso las copia en el banco de registros. Este proceso es independiente de la unidad de forwarding, y por tanto su tiempo no es crítico, lo importante es que no sea tan lento como para permitir que el buffer de reordenamiento se llene.

2.1.3) Casos intermedios: GALS

Dado que tanto los procesadores totalmente síncronos como los totalmente asíncronos presentan problemas, se pensó en combinar las ventajas de una y otra en una nueva alternativa de diseño, y así aparecieron los diseños *GALS* (*Globalmente Asíncronos Localmente síncronos*). Los GALS constan de varios módulos síncronos con relojes independientes, que se comunican entre sí asíncronamente. Los módulos pueden

trabajar a distintas frecuencias entre sí. Y en algunos diseños, el reloj de cada módulo podrá funcionar a su vez a distintas velocidades.

Ventajas de los diseños GALS [5, 6]:

- Inexistencia de reloj global: Cada módulo tiene su propia señal de reloj local De este modo se evita el esfuerzo de diseño producido por llevar la señal con el mínimo desfase y retardo a puntos muy lejanos del sistema. A su vez se reduce también el consumo de potencia y la ocupación de área producidos por la distribución de el reloj.

- Reusabilidad: actualmente estamos entrando en la era de los *Systems On Chip* (*SoC's*), donde distintas empresas producen circuitos, que otros diseñadores pueden comprar para reutilizarlos haciendo diseños de mayor complejidad. De esta forma se reduce enormemente el tiempo de salida al mercado, ya que para producir un chip no es necesario diseñar cada uno de los componentes que lo forman. Además, el mismo circuito puede usarse para crear varios chips distintos.

  Estos circuitos síncronos siguen teniendo atractivo técnico para la industria, ya que ofrecen las siguientes ventajas: Primero, son pequeños luego no se ven tan afectados por los problemas de reloj de los grandes diseños síncronos. Segundo, existen multitud de herramientas de diseño síncrono. Los diseños GALS pueden ofrecer una solución para implementadores de SoC's que buscan buen rendimiento y bajo consumo de potencia. Bloques síncronos pueden ser integrados en un chip con relojes independientes para cada uno y con una interconexión asíncrona entre ellos. Su posible reusabilidad, junto al auge de los chips que son actualmente usados en prácticamente todos los aparatos electrónicos despierta el interés de la industria, de las empresas y los organismos en este tipo de sistemas.

- Inercia: La existencia de herramientas CAD (*Computer Aided Design*) síncronas bastante asentadas y ampliamente conocidas por los diseñadores motiva el desarrollo de circuitos GALS en lugar de totalmente asíncronos (que solucionan los problemas de reloj de los síncronos), ya que éstos no cuentan con herramientas tan eficaces. Por ello la migración hacia los totalmente asíncronos no se dará en un futuro inmediato.

Predicción de futuro:

La asociación industrial de Semiconductores (*Semiconductor Industry Association, SIA*) y el *Electronic Design Automation Industry Council* predicen que en un mismo chip el número de relojes crecerá rápidamente. La SIA predice que en 2007 las técnicas asíncronas serán usadas en muchos diseños [6].

Comunicación asíncrona entre módulos síncronos:

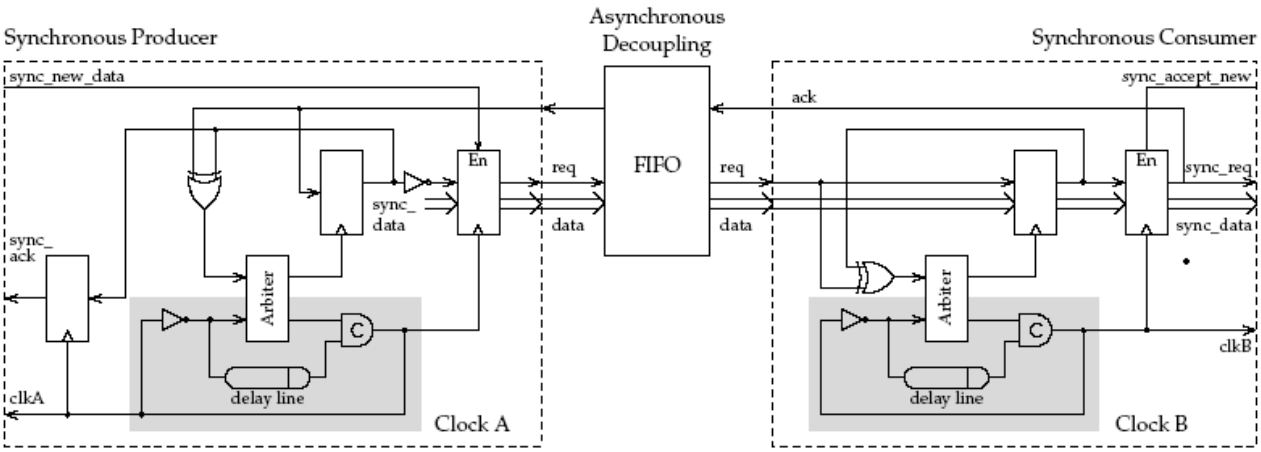
En los sistemas GALS tiene que existir un protocolo de comunicación asíncrona entre los módulos síncronos. Este protocolo es el que se encarga del correcto envío y recepción de los datos entre los dominios de reloj.

Esta comunicación se divide en dos partes del tipo productor-consumidor: una en la que el bloque síncrono pone la información en el canal asíncrono y otra en la que el bloque síncrono la coje del canal asíncrono.

Un problema que surge en ambas comunicaciones entre productores y consumidores es la posibilidad de *metaestabilidad* [6]. La metaestabilidad aparece porque el dato y la señal de reloj llegan al módulo simultáneamente y su principal efecto es que la salida de un sincronizador no es ni un 0 ni un 1 durante un tiempo indeterminado.

Veamos un ejemplo de circuito que evita la metaestabilidad para ambas comunicaciones, basado en el protocolo *Handshake*:

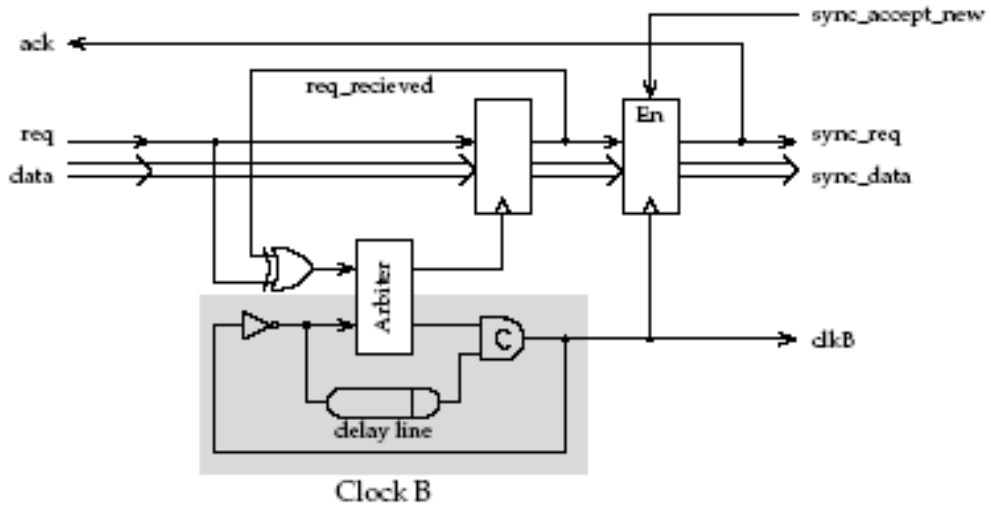
El circuito de comunicación completo entre dos módulos sincronos es [6]:



Donde los rectángulos de líneas discontinuas son los dos bloques síncronos y el canal intermedio, junto con la cola FIFO son el canal asíncrono

Que, dividido en las dos comunicaciones queda:

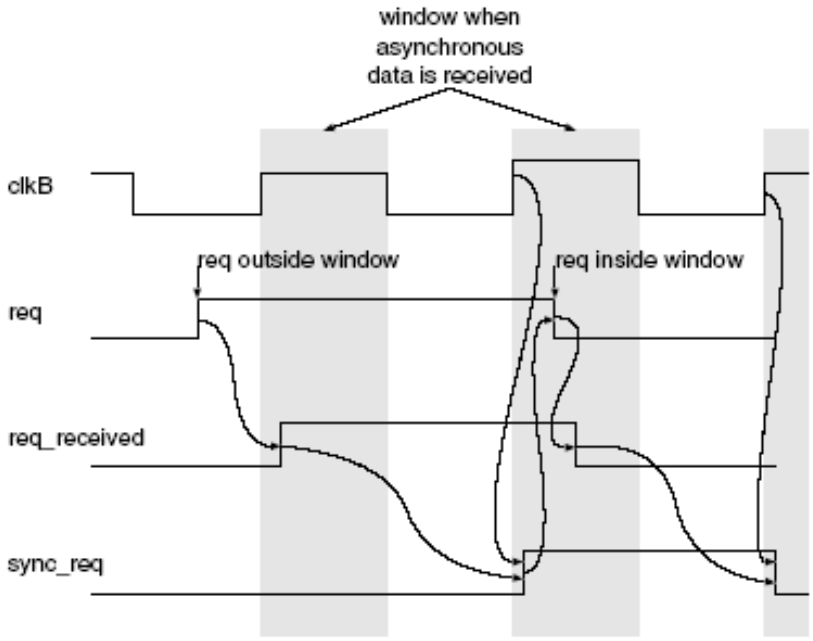
- Comunicación Productor asíncrono (canal) – Consumidor síncrono (el segundo bloque síncrono):



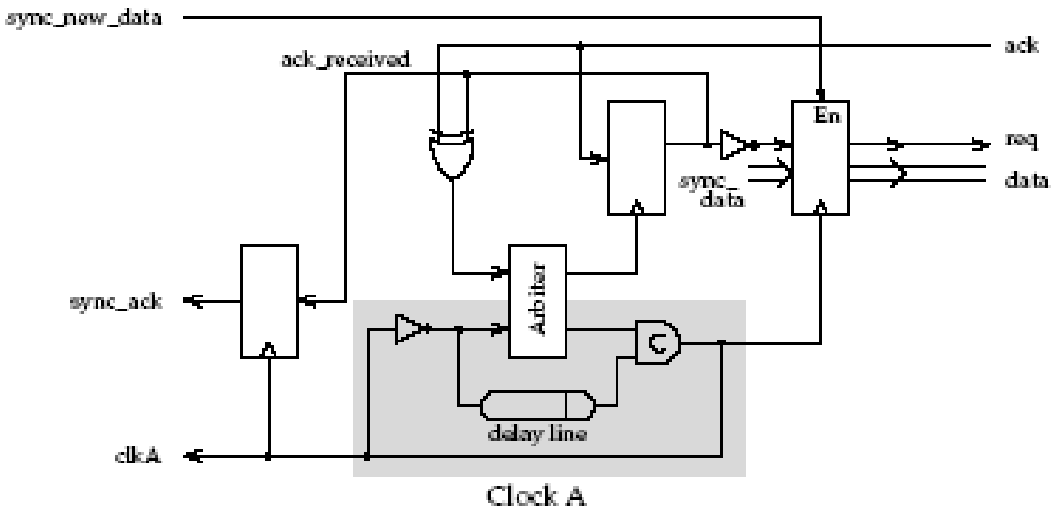
Cuando no se reciben datos del productor asíncrono (el canal), es decir, cuando la señal request (req) está a 0, entonces se genera un proceso oscilatorio que produce una señal de reloj, clkB. Esto se debe a que tras un inversor (que alimenta también a un árbitro), se retrasa en tiempo el valor por una línea de retardo (delay line); después se une con la salida del árbitro mediante una puerta "and" y realimenta a su vez al inversor. De esta forma, si el árbitro saca un 1 siempre, la salida del "not" irá cambiando alternándose por la realimentación por medio de la línea de retardo. Y esa salida alterna es el clkB.

Cuando el reloj está en alta el circuito dará prioridad a los datos procedentes del canal asíncrono. Si llegan estos datos (es decir, llega la señal req) prácticamente a la vez que baja el reloj, el árbitro puede volverse metaestable. Él mismo lo evita, haciendo que el reloj no continúe y que la señal req no se propague hasta que se resuelva la metaestabilidad.

La comunicación punto a punto usa un esquema de señalización de dos fases: el canal envía la señal req, tras recibir el dato asíncronamente, el bloque síncrono responde enviando req_received al árbitro para que al tener req_received y req el mismo valor, éste prohíba que se vuelvan a cargar los datos. Después, al siguiente flanco de subida del reloj del consumidor se sincronizan los datos antes recibidos, enviando un ack (sync_req), avisando así al productor de la correcta recepción, vaciando el canal para permitir futuros envíos, como se ve en el siguiente esquema.

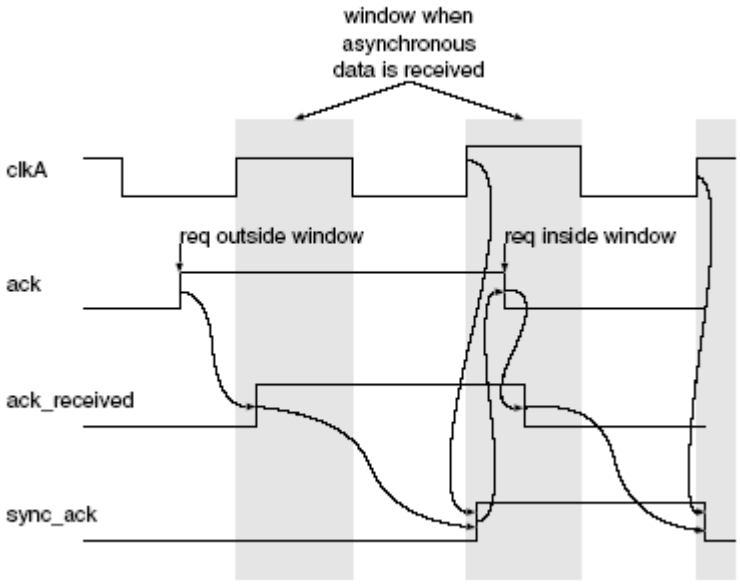


- Comunicación Productor síncrono (el primer bloque síncrono) – Consumidor asíncrono (canal):



En este caso la dificultad estriba en que el productor síncrono necesita saber cuándo el consumidor asíncrono es capaz de recibir datos (por no estar ocupado). Esto se implementa mediante la señal ack.

En cuanto al circuito es prácticamente igual al anterior, pero cambiando las señales req y sync_req por ack y sync_ack. El envío y la recepción de señales son también en dos fases, y tienen el siguiente esquema:



En el que el canal asíncrono envía el ack, que, tras ser detectado por el productor, éste envía ack_received, y al siguiente ciclo de reloj los datos a la vez que pone a "1" sync_ack. Una vez los datos son volcados al canal, baja la señal ack y al ser recibido esto por el productor, hace lo mismo con ack_received, y al siguiente ciclo con sync_ack, terminando así la transmisión.

Estos circuitos de comunicación Bloque síncrono – Canal asíncrono y Canal asíncrono – Bloque síncrono se pueden unir con una cola FIFO para mejorar el rendimiento de la comunicación, como se puede observar en el circuito de comunicación completo (mostrado antes).

Determinismo e indeterminismo:

Cuando las instrucciones se pueden ejecutar y acabar fuera de orden, es posible que en un GALS se de el problema de que en un determinado momento no sepamos qué instrucciones se han ejecutado ya y cuáles no. Esto tiene consecuencias nefastas en la depuración, validación y la prueba, ya que no se puede predecir en qué estado se encuentra el procesador en un determinado momento, ni en qué fase de ejecución se encuentra una instrucción en un instante dado [1].

Así pues, en un GALS a partir de un estado no podemos saber cuál será el estado siguiente, ya que el orden de ejecución de las etapas de las instrucciones depende de diversos factores: el desfase entre los relojes y el retardo en la comunicación entre módulos debido al protocolo utilizado, por ejemplo con el protocolo Handshake hay indeterminismo, pero veremos más adelante que hay un diseño determinista en el que se usa el protocolo Token Ring.

Se puede observar que en un ejemplo de ejecución de cinco instrucciones en un procesador GALS con cuatro etapas (cada una de ellas con un reloj de la misma frecuencia) puede variar el orden de ejecución de cada una de las etapas según los dos factores anteriormente mencionados de la siguiente forma:

Cycle	I1	I2	I3	I4	I5
1	Issue				
2	Read	Issue			
3			Issue		
4			Read	Issue	
5	Exec				
6	Write				
7		Read	Exec		
8			Write	Read	
9					Issue
10					Read
11		Exec		Exec	
12		Write			
13				Write	Exec
14					Write

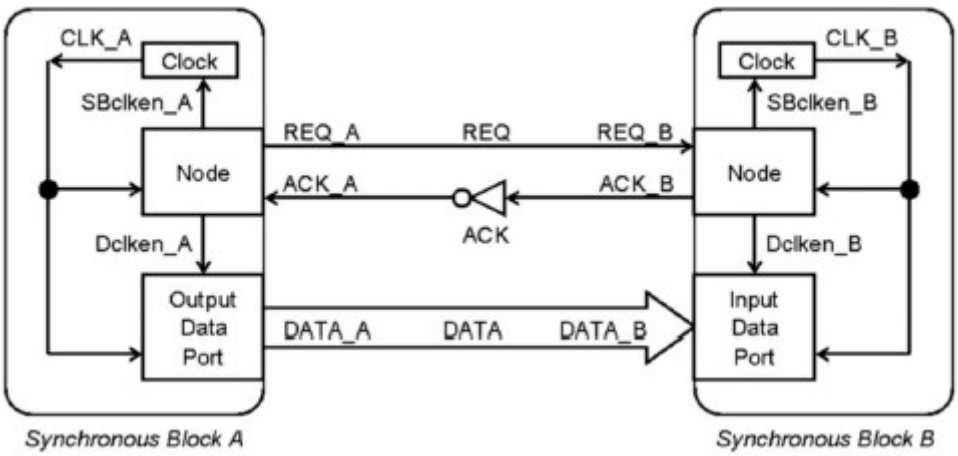
Cycle	I1	I2	I3	I4	I5
1	Issue				
2	Read	Issue			
3			Issue		
4			Read	Issue	
5	Exec				
6	Write				
7		Read			
8			*Exec*	Read	
9			*Write*		
10					*Issue*
11		Exec		Exec	*Read*
12		Write			
13				Write	
14					*Exec*
15					*Write*

Cycle	I1	I2	I3	I4	I5
1	Issue				
2	Read	Issue			
3			Issue		
4			Read	Issue	
5	Exec				
6	Write				
7		Read	Exec		
8			Write	Read	
9					Issue
10				*Exec*	Read
11		Exec		*Write*	
12		Write			
13					Exec
14					Write

En general, en un GALS con *scoreboard* (cuya ejecución es fuera de orden) habrá indeterminismo.

Synchro-Tokens

Para evitar el indeterminismo en sistemas GALS con scoreboard, hay un diseño original que es determinista, la arquitectura "*synchro-tokens*" [1, 7]:



Como se muestra en la figura anterior, un sistema synchro tokens consiste en un conjunto de bloques síncronos (BSs) rodeados de una "envoltura" lógica y conectados con canales de comunicación asíncronos y token rings.

La envoltura lógica consta de: nodos token ring, interfaces asíncronas y relojes que se pueden parar. Uno o más BSs son designados de E/S y se sincronizan a y se comunican con el exterior (el medio), sin intervención de la envoltura lógica.

Canales de comunicación asíncrona transportan datos entre dos bloques síncronos. Dichos canales pueden tener opcionalmente colas FIFO para mejorar el rendimiento. Al final de cada canal nos encontramos con una interfaz asíncrona.

La comunicación de las señales entre dos bloques síncronos se realiza en el anillo token ring que une los nodos de los BSs, usando un protocolo Handshake (de dos fases), en el que se envían las señales req y ack. Cada par de BSs en comunicación tiene un token ring con un nodo en cada envoltura lógica de cada BS.

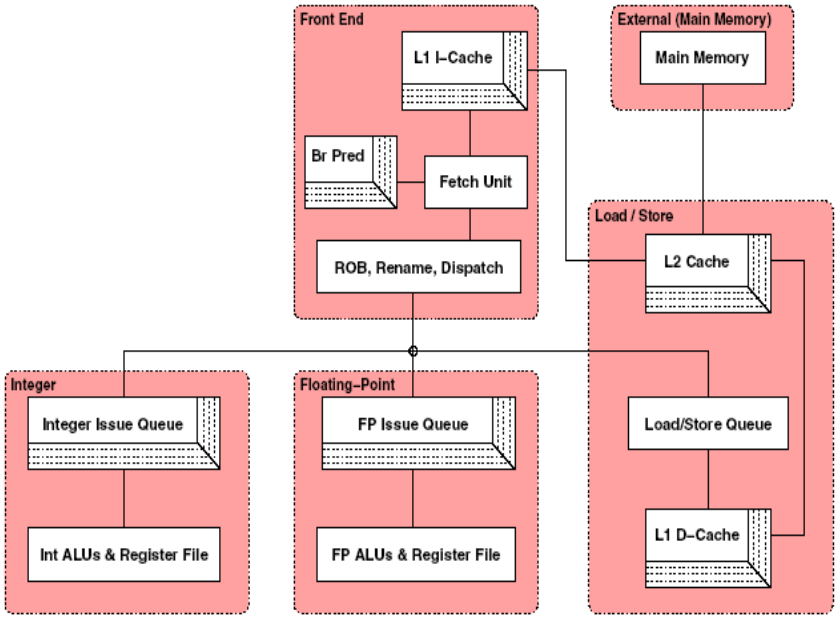
Para evitar el indeterminismo, las envolturas de los BSs deben asegurar que cada transición en cada entrada asíncrona es muestreada en un único ciclo del reloj local. El instante de llegada de la señal asíncrona a la entrada del bloque síncrono no se puede controlar. Por tanto, una transición no debe ser reconocida si se da antes de lo esperado. Y si se da después, el reloj local se parará esperando a que ocurra. Sin embargo, en la práctica no se dispone de la información de cuándo se dará una transición. No obstante, este conocimiento se puede inferir distinguiendo señales de datos (que llevan información en sus niveles lógicos) y señales del protocolo de comunicación (que llevan información en los instantes en que se dan sus transiciones). Ajustando el reloj local con las señales del protocolo (handshake), se consigue ajustar el mismo reloj con los datos que recibe. Así se consigue una recepción de datos determinista.

Cambio dinámico en la frecuencia y en la complejidad en un microprocesador GALS [8]:

Existen aplicaciones que tienen un diseño con una complejidad hardware relativamente baja en favor de una velocidad de reloj mayor, y otras cuya complejidad hardware es alta y en cambio la velocidad de su reloj es menor. Hay estudios que han demostrado que existe una gran variabilidad en el uso de los requisitos del hardware en las diferentes fases de una aplicación dada. Por ello surge la idea de la optimización dinámica de la frecuencia y la complejidad, que trata de optimizar dinámicamente la relación entre la velocidad del reloj y las instrucciones por ciclo.

Se puede aumentar la complejidad de un sistema a cambio de reducir la frecuencia a la que actúa, siempre que este incremento proporcione beneficios en cuanto al número de instrucciones por ciclo, el IPC, que se pueden ejecutar, y que estos beneficios compensen la frecuencia perdida en ese dominio. A continuación describimos un microprocesador GALS MCD que dinámicamente cambia el tamaño de su hardware y ajusta la velocidad de su frecuencia para obtener mayores IPC's.

La estructura de esta arquitectura se puede ver en la siguiente figura en la cual las cajas con múltiples bordes indican las estructuras de tamaño adaptable dinámicamente.

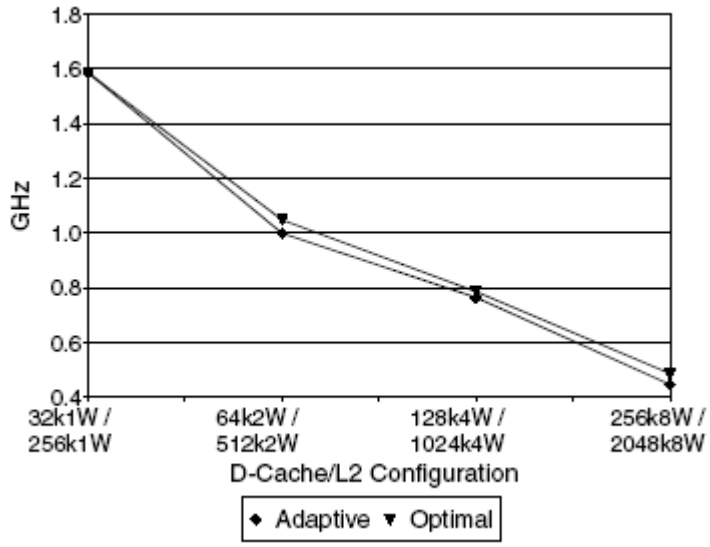


Dominios:

1. *Dominio Load/Store*

Las dos cachés de este dominio son asociativas de 8 vías, y se modifica su tamaño cambiando el grado de asociatividad. La configuración base es el tamaño más pequeño y la mayor velocidad de reloj.

En la figura se observan las frecuencias para una caché óptima y una adaptable.



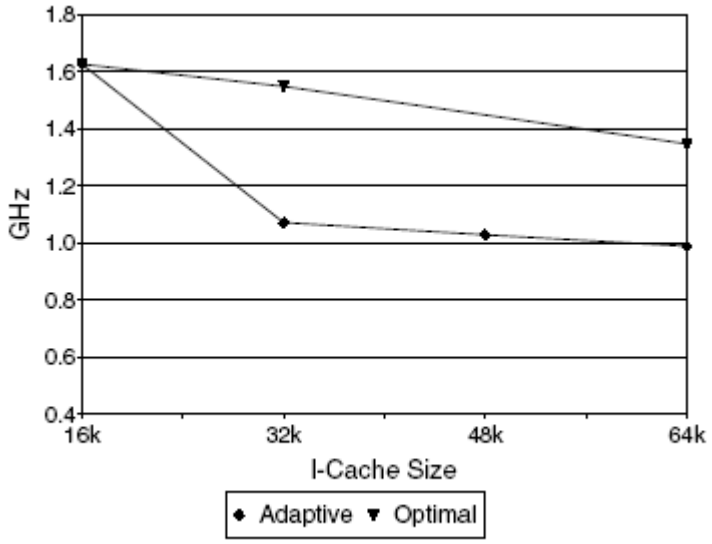
El eje x representa el tamaño y el número de vías de las cachés de datos (L1) y la L2. El eje y representa la frecuencia.

Se observa que la relación de frecuencias entre la caché óptima y la adaptable es aproximadamente de un 5%, luego el comportamiento de una caché adaptable se asemeja mucho al de una óptima.

2. *Dominio Front-End*

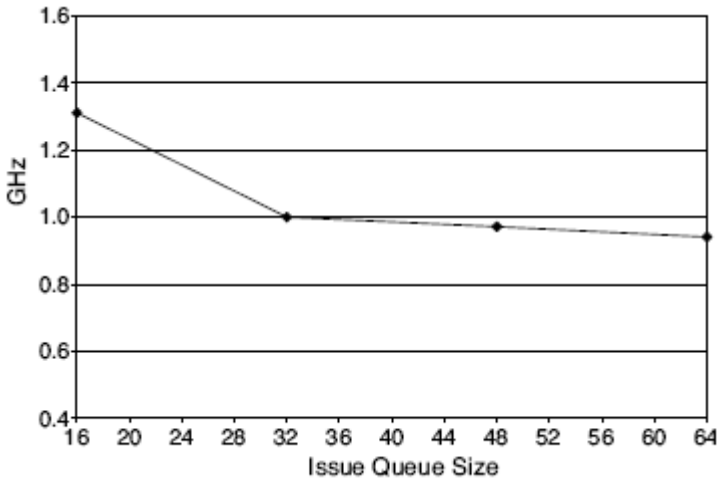
En este dominio tanto la caché nivel 1 de instrucciones como el predictor de saltos son adaptables, y siempre cambian de tamaño a la vez. El predictor de saltos es un diseño híbrido con una tabla de historia global, una de historia local y un selector para elegir cuál de las dos usar. El tamaño de la caché se aumenta incrementando su grado de asociatividad (de 1 a 4) y el del predictor de saltos aumentando el número de bits para sus tablas de historia global y local.

Se observa en la siguiente gráfica una comparativa entre una caché óptimamente configurada y una caché adaptable. Como se puede ver, hay una gran diferencia en la frecuencia, del orden del 31%.



3. *Dominios Integer & Floating-Point*

Las colas de lanzamiento pueden variar de tamaño entre 16 y 64 entradas, con incrementos de cuatro en cuatro. Pero se sufre una importante disminución de la frecuencia al cambiar de una cola con 16 entradas a una con 64, este efecto de la frecuencia causa que la cola de lanzamiento de 16 entradas sea la mejor opción, como se puede ver en la siguiente gráfica.



Algoritmos de control para cachés y colas de lanzamiento

Tanto las cachés como las colas de lanzamiento necesitan algoritmos de control para soportar la adaptación en las distintas fases de la aplicación.

Cachés adaptables: recopilan estadísticas, y calculan el número de aciertos y de fallos que se darían en un lapso de tiempo para cada configuración, con lo que eligen la configuración a utilizar en un intervalo de tiempo dado, en base a la configuración que ha causado el mínimo coste de accesos totales en el intervalo anterior.

Colas de lanzamiento adaptables: se introduce un algoritmo para medir el paralelismo a nivel de instrucción (ILP) en la aplicación que se está ejecutando en ese momento.

Algoritmo: se calcula el número de las dependencias M (si una instrucción usa como operando fuente el operando destino de la instrucción anterior se aumenta M en uno), se toma N como el tamaño de la cola (16, 32, 48...). En este momento se estima el paralelismo como M/N. Desde luego, esta división no sirve ya que el numerador es siempre igual, y la comparación sería simplemente entre denominadores, lo cual no tiene mucho sentido, con lo que el algoritmo hace corresponder a cada una de estas 4 estimaciones con la frecuencia, las compara y determina qué tamaño de cola de lanzamiento hubiera dado mejores resultados, en el pasado reciente, para conseguir un ILP más efectivo.

Problemas GALS:

En un estudio de 2003, se realizó un entorno de simulación preciso para estudiar el impacto del asincronismo en una arquitectura de procesador superescalar GALS. Se realizó suponiendo que todos los módulos tenían la misma frecuencia, no modificable en tiempo de ejecución.

Los resultados muestran, como se esperaba, que los diseños GALS producen una caída en el rendimiento, pero la eliminación del reloj global no lleva a reducciones de potencia drásticas. Los diseños GALS son intrínsecamente menos eficientes al

compararlos con arquitecturas síncronas. Sin embargo, la flexibilidad debido al uso de relojes locales permite conseguir mayor efectividad de otras técnicas de conservación de energía, como "*dynamic voltage scaling*". Los cálculos, para un procesador GALS de 5 dominios de reloj, dan una caída de rendimiento de entre un 5 y 15%, mientras que el consumo de potencia se reduce en un 10% en media.

Si bien, se espera que al permitir que los módulos tengan distintas frecuencias entre sí, y que a su vez puedan cambiar su frecuencia en tiempo de ejecución, se reduzca aún más el consumo de potencia, y se oculte gran parte de esa pérdida de rendimiento.

2.2) Simultaneous Multithreading:

2.2.1) Introducción

Los superescalares son procesadores que pueden lanzar más de una instrucción por ciclo. Para que ofrezcan un buen rendimiento, han de eliminar las dependencias que surgen debido a las instrucciones que se ejecutan concurrentemente. Los superescalares con ejecución fuera de orden de hoy en día, usan técnicas como el renombramiento de registros y la planificación dinámica para eliminar riesgos creados por la reutilización de los registros, y para ocultar grandes latencias de ejecución resultantes de los fallos de la caché de datos y de las operaciones en punto flotante. Sin embargo, el método básico de búsqueda y envío secuencial es todavía el modelo computacional subyacente. Por este motivo, el desarrollo de los superescalares está limitado por los fallos de predicción de saltos y de caché.

Debido a que estos fallos ocurren frecuentemente, el rendimiento obtenido está bastante por debajo del pico del ancho de banda de un superescalar. Idealmente necesitamos un suministro no interrumpido de instrucciones para aumentarlo. Aún entonces, hay otras complejidades que tienen que ser superadas.

El renombramiento de registros requiere una comprobación de dependencias entre instrucciones del mismo bloque, y múltiples puertos de lectura en la tabla de renombramiento. Esta lógica incrementa su complejidad a medida que aumenta la anchura de la etapa de renombramiento.

Se necesitan muchas instrucciones para poder encontrar un subconjunto de ellas lo suficientemente independientes como para dar a las unidades de ejecución una utilización completa. La lógica de lanzamiento tiene que identificar instrucciones independientes rápidamente, cuyas entradas estén preparadas, y lanzarlas para que se ejecuten.

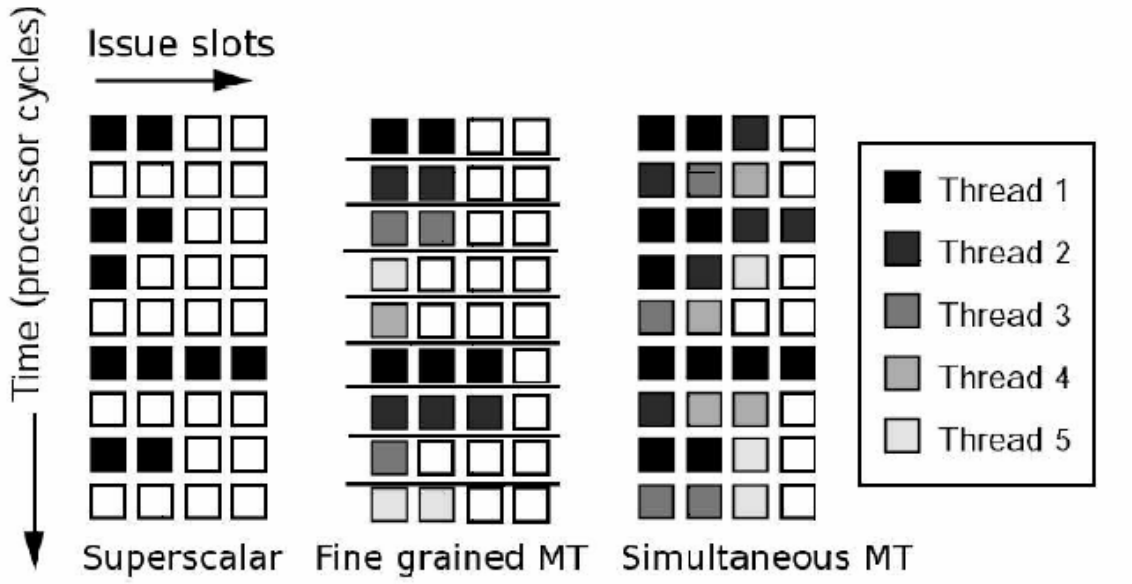
La única manera de mejorar el diseño de los procesadores es aumentar las capacidades computacionales del procesador. En general, esto significa incrementar el paralelismo en todas sus formas posibles. Los superescalares, por ejemplo, en teoría pueden ejecutar hasta ocho instrucciones por ciclo, aunque en la práctica sólo consiguen ejecutar seis como mucho, ya que las aplicaciones actuales tienen un bajo paralelismo a nivel de instrucción.

Existen varios tipos de pérdidas de rendimiento debidas a que no se explota eficientemente el paralelismo [16]. Una de estas es la horizontal, surge cuando se lanzan menos instrucciones de las posibles. Esto se debe al poco paralelismo que hay a nivel de instrucción.

La pérdida vertical ocurre cuando en un ciclo ninguna instrucción está utilizando los recursos hardware. Esto puede ser debido a una gran latencia de las instrucciones (debido quizá a un acceso a memoria) que inhibe el lanzamiento posterior de instrucciones.

La técnica llamada multithreading fue pensada precisamente para poder aumentar este paralelismo a nivel de instrucción. Un procesador multihilo es capaz de ejecutar instrucciones de diferentes hilos de ejecución simultáneamente y de esta forma es más sencillo encontrar instrucciones que no tengan dependencias entre sí.

Hay varios tipos de arquitecturas multihilo: multihilo de grano fino, de grano grueso y Simultaneous Multithreading [17]. Se muestra a continuación una representación gráfica en la siguiente figura:



La figura 1.a muestra una secuencia posible de ejecución de un superescalar. Como en todos los superescalares, se ejecuta un único programa, o hilo, del que se intentan encontrar múltiples instrucciones para lanzar en cada ciclo. Cuando no se pueden encontrar dichas instrucciones, no hay posibilidad de rellenar los huecos, lo que causa tanto gasto horizontal como vertical.

En la figura 1.b se puede observar una posible secuencia de ejecución de instrucciones de una arquitectura multihilo de grano fino. Los procesadores multihilo contienen hardware de estado (contador de programa y registros) para varios hilos. Dado un ciclo, el procesador ejecuta instrucciones de uno de los hilos. En el ciclo siguiente, cambia a un contexto diferente y ejecuta instrucciones del nuevo hilo. Pero

aun así, este tipo de procesadores siguen limitados por el paralelismo de instrucciones en un único hilo.

En una arquitectura multihilo de grano grueso, en cada ciclo se ejecutan instrucciones de un hilo, en el momento en que este hilo sufre una parada, debida quizás a un fallo de caché, es cuando se da un cambio de contexto y entra a ejecutarse un nuevo hilo.

La figura 1.c muestra como un procesador SMT puede seleccionar múltiples instrucciones de múltiples hilos cada ciclo desde un único pipeline. Un diseño de tipo SMT explota el paralelismo a nivel de instrucción al seleccionar instrucciones de varios hilos. El paralelismo a nivel de hilo, puede venir de programas multihilo, paralelos o programas individuales que transcurren en una carga de trabajo de multiprogramación.

El procesador asigna dinámicamente los recursos de la máquina a las instrucciones, proporcionando una mayor oportunidad para una mayor utilización hardware. Los procesadores SMT heredan de los superescalares la habilidad de lanzar múltiples instrucciones cada ciclo, y de los multihilo, la posibilidad de coexistencia de varios hilos en ejecución.

Interés actual en los multihilo:

Dos tendencias recientes han aumentado el interés en multihilo [18]:

1.- La tecnología de fabricación de conductores es ahora capaz de producir microprocesadores superescalares cuyo pico del rendimiento potencial está más allá del rendimiento que puede ser extraído de la mayoría de las aplicaciones de un solo hilo.

2.- El sistema operativo, el compilador y el soporte de lenguaje para multihilo se están convirtiendo en algo más difundido como por ejemplo Windows NT y JAVA.

Ventajas de los procesadores SMT:

SMT es una forma de multihilo propuesta por Tullsen et al. [15]. Permite compartir recursos de grano fino en un procesador superescalar dinámico con ejecución fuera de orden. Debido a esto, el rendimiento total puede ser mejorado significativamente con respecto a un procesador monohilo. Esto hace que las ganancias en velocidad y rendimiento sean mayores.

El hecho de que múltiples instrucciones de múltiples hilos se puedan ejecutar en el mismo ciclo, proporciona una gran flexibilidad que permite al procesador SMT apantallar paradas en un hilo al ejecutar instrucciones de otros. Pero esta flexibilidad tiene un coste, el archivo de registros y las tablas de renombramiento han de ser agrandadas para acomodar los registros de la arquitectura de los hilos adicionales. Esto puede incrementar el tiempo de ciclo de reloj y/o el número de etapas del pipeline.

Relativas al consumo de potencia:

El consumo de potencia en los microprocesadores se está convirtiendo cada vez más en un factor importantísimo que afecta a las decisiones de diseño, debido a la necesidad de reducir tanto el consumo de potencia propiamente dicho (especialmente en los portátiles), como disminuir la densidad del mismo: menor consumo por unidad de área.

En los procesadores SMT el consumo de potencia no aumenta de forma proporcional al incremento del rendimiento, sino que lo hace en menor medida. Esto proporciona un cierto atractivo a los SMT en el consumo de potencia por instrucción. Se debe a:

- El paralelismo extra que proporciona: el procesador depende mucho menos fuertemente de la especulación; es por esto que gasta menos recursos en las instrucciones especuladas y lanzadas y de las cuales nunca se hace commit.

- Dado que un SMT proporciona más paralelismo y puede ejecutar instrucciones de múltiples hilos en cada ciclo, utilizará de forma más eficiente los recursos, gastando así menos potencia en hardware inutilizado.

Relativas a la complicación del hardware:

Otra ventaja de estos procesadores es que no necesitan hardware especial para programar instrucciones de los diferentes hilos en las unidades funcionales. El hardware de la programación dinámica en los superescalares es todavía funcionalmente capaz de programar SMT. Con el renombramiento de registros se consiguen eliminar conflictos que se producen al referenciar repetidamente el mismo registro, mapeando los registros de la arquitectura específicos en los registros hardware; el procesador entonces lanza instrucciones (después de que sus operandos hayan sido calculados o cargados de memoria) sin considerar al hilo. Este mínimo rediseño tiene dos consecuencias importantes:

La primera es que dado que la mayoría de los recursos hardware están todavía disponibles para la ejecución de un único hilo, SMT proporciona buen rendimiento para los programas que no pueden ser paralelizados.

La segunda, debido a los cambios para permitir SMT son mínimos, la transición comercial de los superescalares a los procesadores SMT no causa muchos problemas. Los diseñadores de hardware se pueden centrar en construir un superescalar de un solo hilo muy rápido y añadir después la capacidad de multihilo.

Formas de implementar el SMT:

Hay dos formas de implementar SMT, el paralelismo a nivel de hilo (TLP) y la precomputación especulativa (SPR). Normalmente se han utilizado de forma separada, aunque se pueden obtener beneficios si se usa una combinación de ambas [19].

El paralelismo a nivel de hilo (TLP) [20], equivale a hacer un programa paralelo, ya sea manualmente o con la asistencia del compilador, y asignar diferentes hilos a

diferentes contextos hardware de ejecución en el procesador. TLP puede acelerar la ejecución de un programa al usar las múltiples unidades de ejecución y explotando el ILP, además oculta la latencia de memoria.

La precomputación especulativa (SPR) puede hacerse de varias formas, la más común es que un hilo ejecute la mayor parte de la computación, mientras que otro hilo, llamado el hilo ayudante, se encargue de ejecutar los load's delincuentes, y las instrucciones que producen sus direcciones. Estos load's son aquellos que tienen posibilidad de sufrir un fallo de caché con elevada latencia. SPR se consigue lanzando el hilo ayudante mientras el hilo principal se está ejecutando.

SPR sólo se ocupa de la latencia en memoria y no explota el paralelismo, pero en cambio consigue ganancias en programas que tienen patrones irregulares de acceso a memoria.

En cambio, TLP tiene algunas ventajas en arquitecturas en las que los hilos tienen suficientes recursos para ejecutar un gran ILP, pero también tiene sus limitaciones, como por ejemplo los conflictos que surgen a la hora de usar recursos compartidos.

El objetivo de integrar SPR y TLP, es utilizar mejor los contextos de ejecución del procesador y lograr una mayor velocidad, este aumento en la velocidad se consigue gracias a la ocultación de las latencias de memoria, y a utilizar una ejecución paralela. Programas con partes secuenciales y partes paralelas, se benefician directamente de este enfoque.

Eficiencia en la cola de lanzamiento:

SMT proporciona un gran rendimiento debido a la ejecución simultánea de las instrucciones entre múltiples hilos. Para maximizar el rendimiento, se debe adoptar un equilibrio entre el paralelismo y la frecuencia de reloj en el diseño de estos procesadores. Estructuras muy complejas pueden hacer que la frecuencia de reloj se degrade, lo que lleva a una disminución en el rendimiento. Una consideración adicional es la disipación de potencia, que ha aparecido como una de las limitaciones más importantes en el diseño de microprocesadores.

La cola de lanzamiento puede ser también una fuente importante de esta disipación de potencia. Por ejemplo, en el Alpha 21464, es la unidad funcional que más potencia consume [21]. También tiene una alta densidad de consumo de potencia, lo que puede llevar a problemas de calentamiento.

El uso de SMT hace que sea más importante el utilizar una cola más rápida y más eficiente en cuanto al consumo de potencia por dos razones:

- La primera es que la ventana de instrucciones de un SMT puede soportar instrucciones de varios hilos para aumentar su rendimiento, esto hace que la cola de lanzamiento sea mayor ya que la ventana de instrucciones en el SMT es muy grande.

- La segunda es que en un procesador SMT, la inhabilitación de un único hilo para utilizar la cola de lanzamiento (por ejemplo, un fallo de caché), puede ser obviada rellenando la cola con otros hilos. El resultado es una utilización mayor de la cola de lanzamiento comparado con una máquina monohilo.

Un problema importante de la cola en SMT es la obstrucción del lanzamiento. Esto sucede cuando hay instrucciones que residen en la cola durante muchos ciclos antes de ser lanzadas. Estas instrucciones ocupan entradas que en las que podrían estar otras a las que les falta menos tiempo para ser lanzadas.

Hay tres causas para la aparición de este tipo de problemas:

- Una es la gran latencia de las instrucciones, como por ejemplo la división o multiplicación en punto flotante, o las instrucciones load que producen fallo de caché.

- Otra razón son las cadenas de dependencias de datos que retrasan el lanzamiento de las instrucciones (por supuesto, la longitud de estas cadenas viene dada por el primer factor, la latencia de algunas instrucciones).

- La tercera causa es la ausencia de unidades funcionales libres, es decir, instrucciones cuyos operandos están disponibles pero que no pueden entrar a ejecutar porque no hay suficientes recursos disponibles.

Las instrucciones que residen en la cola de lanzamiento dependen directamente de la política de fetch. Es por esto que la elección de una buena política es importante para que se pueda extraer el máximo paralelismo de aquellas instrucciones que se encuentran en la cola.

Políticas de Fetch:

ICOUNT:

Tullsen exploró una variedad de políticas de fetch de SMT que asignaban prioridades de fetch a los hilos en base a varios criterios. Una política se determina en base al ICOUNT [22], en la que se asigna la prioridad a los hilos en base al número de instrucciones que tienen en las etapas del pipeline.

A los hilos que tienen menos instrucciones se les da la mayor prioridad para fetch, la razón es que estos hilos pueden estar adelantando más trabajo que otros, y para prevenir que un hilo obstruya la cola de lanzamiento y de esta forma proporcionar un conjunto de hilos en la cola de lanzamiento para incrementar el paralelismo. Un esquema de ICOUNT tiene dos parámetros, el número de hilos y el número de instrucciones. El primero indica el máximo número de hilos en los que hacer búsqueda en cada ciclo, mientras el segundo denota el máximo número de instrucciones para traer por cada hilo.

IPQOSN:

Un esquema diferente es aquel que se presenta en [23], en él las instrucciones que están al principio de la cola de lanzamiento, son las que llevan más tiempo (las más antiguas), por lo que tienen una mayor probabilidad de ser las causantes de la obstrucción de la cola. Es por este motivo por lo que esta política favorece aquellos hilos cuyas instrucciones están distribuidas más hacia el final de la cola (las más recientes).

Pero no proporciona una ventaja significante ni en cuanto al rendimiento ni en cuanto a la ocupación de la cola de lanzamiento sobre ICOUNT. Además tiene una complejidad añadida, que es la de seguir la pista de instrucciones dentro de la cola de lanzamiento (ICOUNT, por el contrario, sólo requiere contadores por cada hilo que se incrementan en la etapa de búsqueda y se decrementan en la etapa de lanzamiento).

Las siguientes tres políticas (UCG, DG y PDG) fueron propuestas por A. El-Mourse y D.H_ Albonesi en [24].

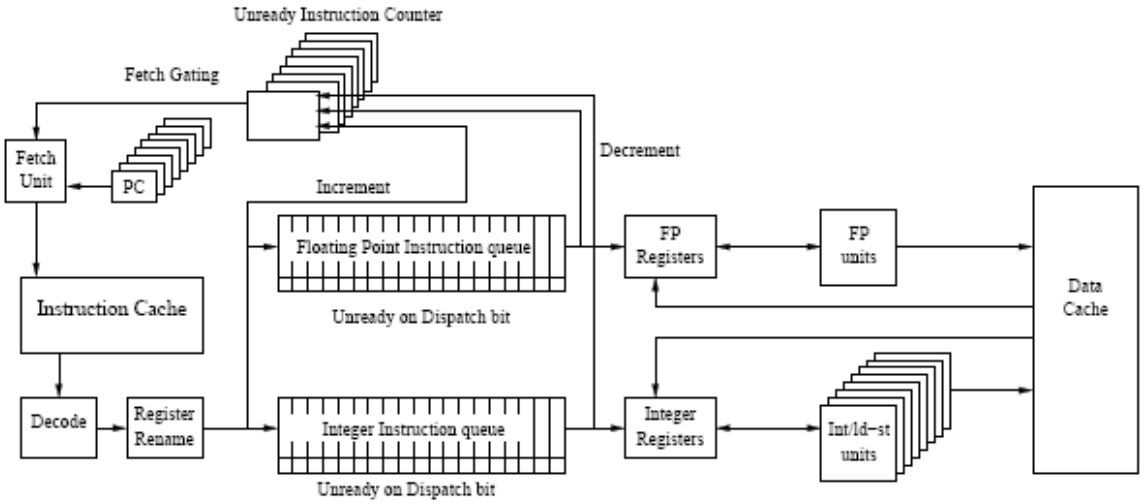
Contador de gating no preparado. UCG

La principal causa de la parada de la cola de lanzamiento viene dada por las instrucciones no preparadas, es decir, aquellas que ocupan la cola de instrucciones mientras esperan a que sus operandos estén disponibles. Esta política UCG, intenta limitar el número de instrucciones no preparadas en la cola para un hilo dado.

La simplificación propuesta por Albonesi opera como sigue: A cada instrucción se le asociada un tag, que indica si dicha instrucción está preparada o no en el momento en que se envía a alguna cola de lanzamiento (bit de unready on dispatch). Por cada hilo se tiene también un contador de las instrucciones que tiene residiendo en alguna cola pero que no están preparadas para la ejecución.

Cuando se hace dispatch de una instrucción de un hilo, si hace falta marcar el bit de no preparado, el contador de instrucciones no preparadas de este hilo se incrementa en uno, y en el momento en que una instrucción de la cola se lanza a las unidades funcionales, si ésta tenía el tag de no preparado marcado, se decrementa dicho contador.

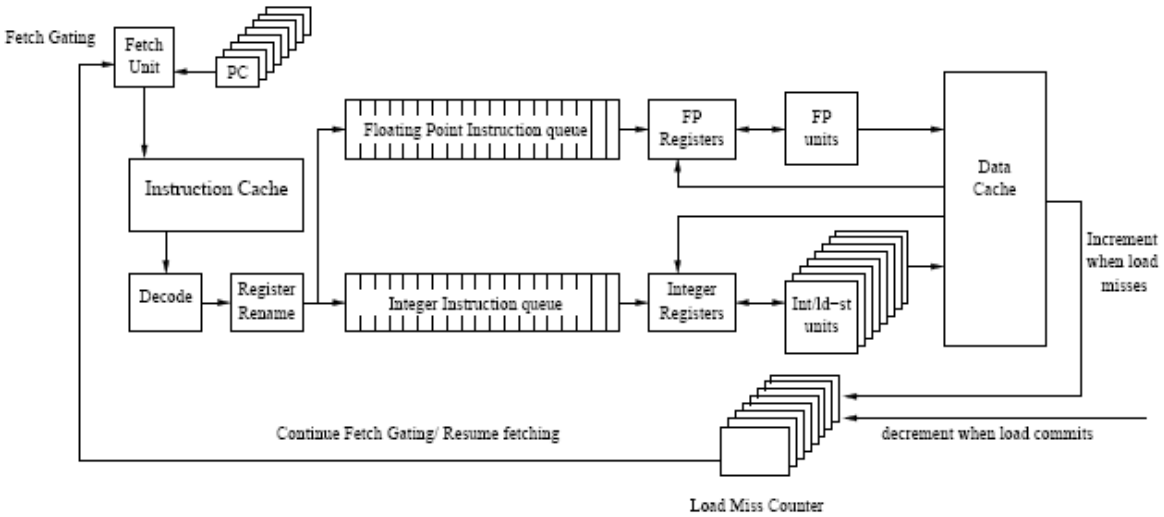
Cualquier hilo cuyo contador de instrucciones no preparadas supere cierto umbral predeterminado, se bloquea, de forma que no se realiza búsqueda de instrucciones de este hilo.



Gating de fallo de datos. DG.

En este esquema, no se ejecuta la búsqueda de instrucciones para cualquier hilo que tenga más de n fallos de caché pendientes. Hay un contador por hilo que se incrementa en los fallos que genera un load, y se decrementa cuando el load se resuelve.

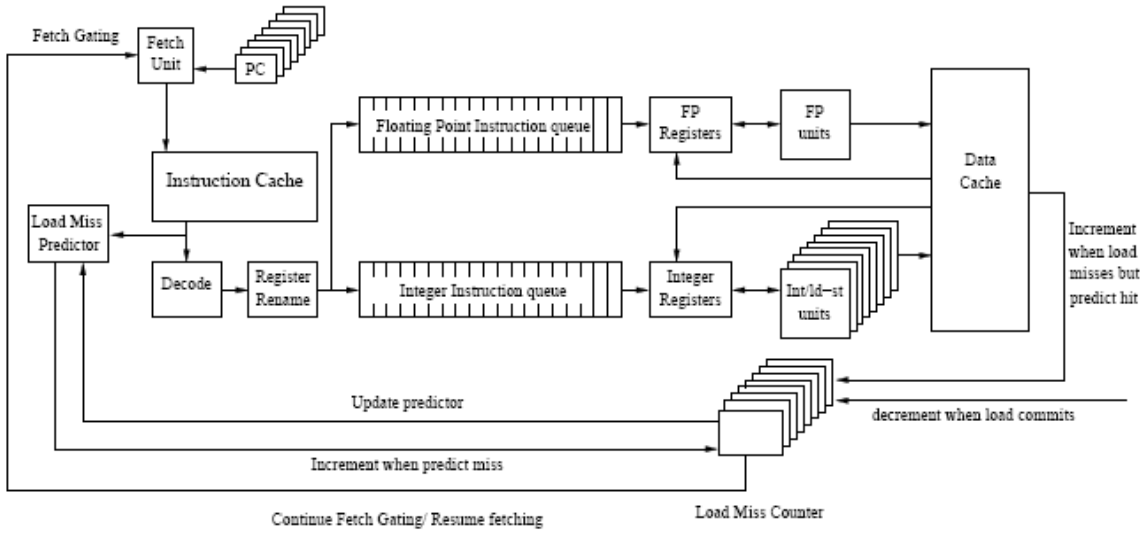
No se realiza búsqueda de instrucciones de un hilo mientras su contador sea mayor que n.



Gating de datos predictivo. PDG.

Una cuestión importante con respecto a la política anterior es que hay un retardo entre la detección de un fallo de caché en la etapa de ejecución y la actualización del valor del contador del hilo. Este retardo puede ocasionar que instrucciones dependientes del load que se está ejecutando se sitúen en la cola antes de que el contador del hilo sea actualizado, por lo que bloquea la cola. Usando esta política, se intenta reducir este retardo prediciendo un fallo de caché en cuanto el load finaliza su ejecución.

Se usa el mismo contador que en el caso anterior pero se incrementa en dos ocasiones diferentes: una cuando se predice que un load va a causar un fallo, y otra cuando se predice un acierto de caché pero la predicción fue incorrecta y realmente hay un fallo. El contador se decrementa cuando el load se resuelve.



Ejemplos de SMT en procesadores:

Pentium IV: El procesador Pentium IV (desarrollado por Intel) es el primer procesador de propósito general disponible comercialmente que implementa SMT. La tecnología de HyperThreading (marca registrada de Intel para la microarquitectura SMT del Pentium IV) hace que el usuario perciba un único procesador como dos lógicos [24].

Para conseguir esto hay una copia del estado de la arquitectura para cada procesador lógico, y éstos comparten un conjunto de recursos físicos de ejecución. La técnica de HyperThreading consigue un incremento del rendimiento con un coste mínimo, debido a que no aumenta mucho el tamaño del circuito. Los procesadores lógicos comparten prácticamente todos los recursos del procesador físico, incluyendo cachés, unidades de ejecución, predictores de saltos, lógica adicional de control y replicación de unos pocos recursos del procesador. Aunque este aumento del área del circuito es pequeño, el incremento en cuanto a complejidad es importante.

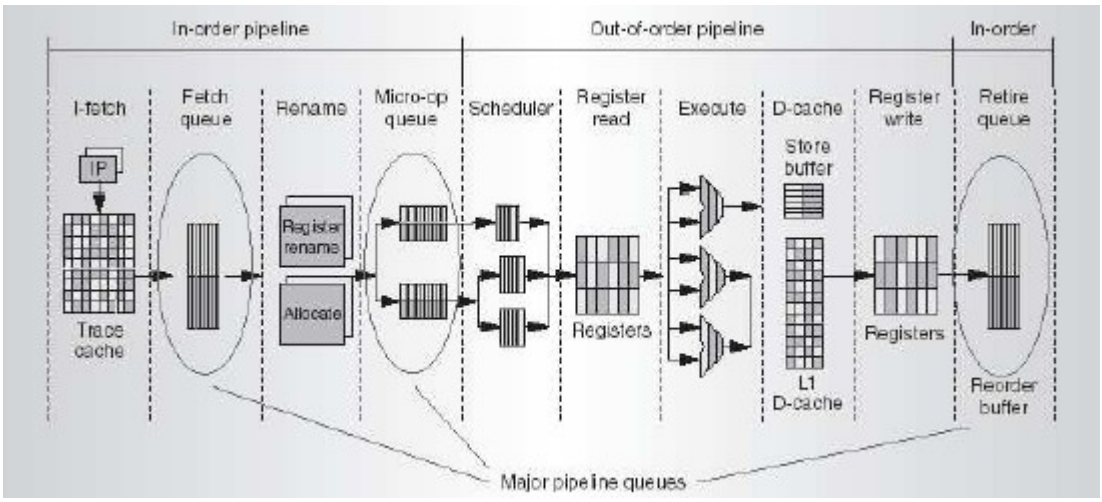
La política de compartimiento de recursos tiene un gran impacto en el rendimiento. Intel elige entre varios esquemas posibles, dependiendo del tipo de estructura, que son:

Particionamiento: dedicar los mismos recursos a cada procesador lógico. Es una buena elección para las colas principales del pipeline, las cuales están llenas la mayor parte del tiempo.

Umbral: compartimiento de recursos con un límite máximo. Es más apropiado para pequeñas estructuras donde la utilización de los recursos es a ráfagas.

Compartimiento completo: La forma de compartir los recursos es muy flexible, no hay restricciones. Es una buena elección para estructuras grandes donde la carga de trabajo es variable, y un procesador lógico no pueda hacer que el otro sufra de inanición, un ejemplo de estas estructuras son las cachés del procesador.

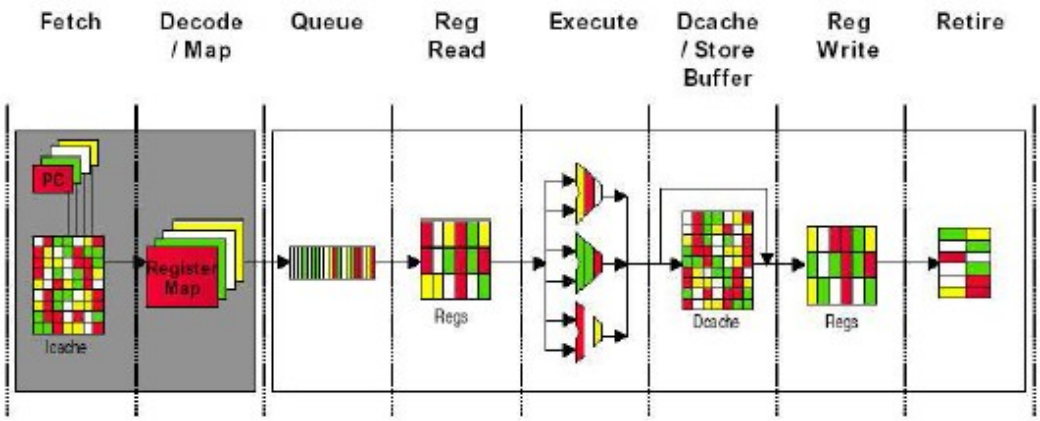
En la siguiente figura se muestra el pipeline del Pentium IV.



Alpha 21464: Creado en 1999 por Compaq (acabó abandonando el proyecto y vendió la tecnología del procesador Alpha a Intel) [24]. Implementa SMT añadiendo el hardware suficiente para mantener 4 contadores de programa diferentes y seleccionar un único hilo cada ciclo del que hacer fetch de la caché de instrucciones. La lógica de lanzamiento fuera de orden es libre de seleccionar instrucciones preparadas de cualquiera de los 4 hilos simultáneos.

Alpha 21464 sólo duplica los contadores de programas y las tablas de renombramiento de registros. El resto de los recursos se comparten, como el archivo de registros (aunque se incrementa su tamaño), la cola de instrucciones, el primer y segundo nivel de caché, los TLB y el predictor de saltos.

En la siguiente figura se muestra el pipeline del alpha 21464 donde se pueden ver bien los recursos compartidos y los que tienen que ser duplicados.



Blue Gene: IBM comenzó en 1999 a desarrollar una máquina paralela con el objetivo de avanzar en la investigación de proteínas. Esta máquina, llamada Blue Gene, tiene una arquitectura muy compleja, construida especialmente para las aplicaciones de supercomputadores. La máquina consiste en un millón de procesadores que funcionan a un gigaFLOP/s. Cada uno de estos procesadores, utiliza simultaneous multithreading y está formado por 4 hilos. La comunicación dentro de cada procesador se realiza mediante memoria compartida, y entre procesadores mediante paso de mensajes.

3) Arquitectura SMT-GALS

3.1) Introducción:

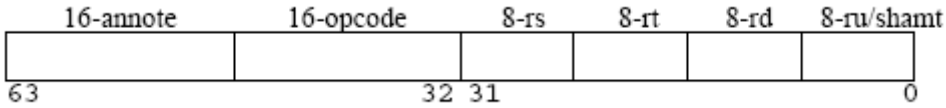
Dada la ventaja en el rendimiento que proporciona el lanzamiento múltiple de instrucciones, y debido a que en el mismo hilo las instrucciones tienen muchas dependencias entre sí, una arquitectura SMT (Simultaneous Multithreading) ofrece un alto rendimiento al poder lanzar en el mismo ciclo varias instrucciones de diferentes hilos. Por otro lado, como ya hemos visto, una arquitectura completamente síncrona tiene problemas de distribución de reloj que podría eliminarse si se usara una arquitectura de tipo GALS

En nuestra investigación partimos de una arquitectura SMT-GALS base, unión de las arquitecturas SMT de Ali El Moursy y la GALS de David H. Albonesi. Esta arquitectura no se ha probado con anterioridad, y sus parámetros no se han ajustado para aprovechar las ventajas que ofrece un diseño SMT. Nuestro objetivo consiste en sintonizar tanto la parte SMT como la GALS. Para realizar el ajuste SMT buscamos que aumente el rendimiento en una ejecución con varios hilos (dos en nuestro caso). Una vez conseguido este ajuste, para la sintonización GALS estudiaremos cuáles son los dominios de reloj óptimos en los que tendríamos que dividir el sistema total. Las simulaciones de este estudio se realizarán usando una variante del simulador SimOutorder, del conjunto de herramientas de simulación SimpleScalar, realizado por Sonia López Alarcón (DACYA) integrando las versiones SMT y GALS.

3.2) Repertorio de instrucciones

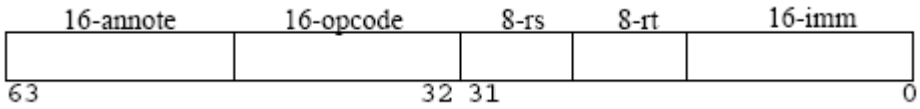
El repertorio de instrucciones que usa la arquitectura que estudiamos es, por tanto, el mismo que el del SimpleScalar. Consta de 111 instrucciones, que se clasifican en instrucciones de control (12), load/store (34), aritmética entera (31), aritmética en coma flotante (28) y otras (nop, syscall, break, lui, mfc1, mtc1). Son instrucciones de 64 bits y tienen los siguientes formatos:

Formato instrucciones tipo R (aritmético-lógicas):



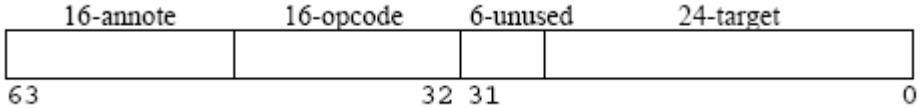
annote: bits de anotación
opcode: código de operación
rs: registro fuente 1
rt: registro fuente 2
rd: registro destino
ru/shamt: se usa para indicar el número de bits a desplazar en las instrucciones de desplazamiento aritmético o lógico.

Formato instrucciones tipo I (con memoria, aritméticas (con operando inmediato) y de salto condicional):



annote: bits de anotación
opcode: código de operación
rs: registro fuente
rt: registro fuente (store, salto condicional), registro destino (load).
inm: operando inmediato

Formato instrucciones tipo J (salto incondicional):

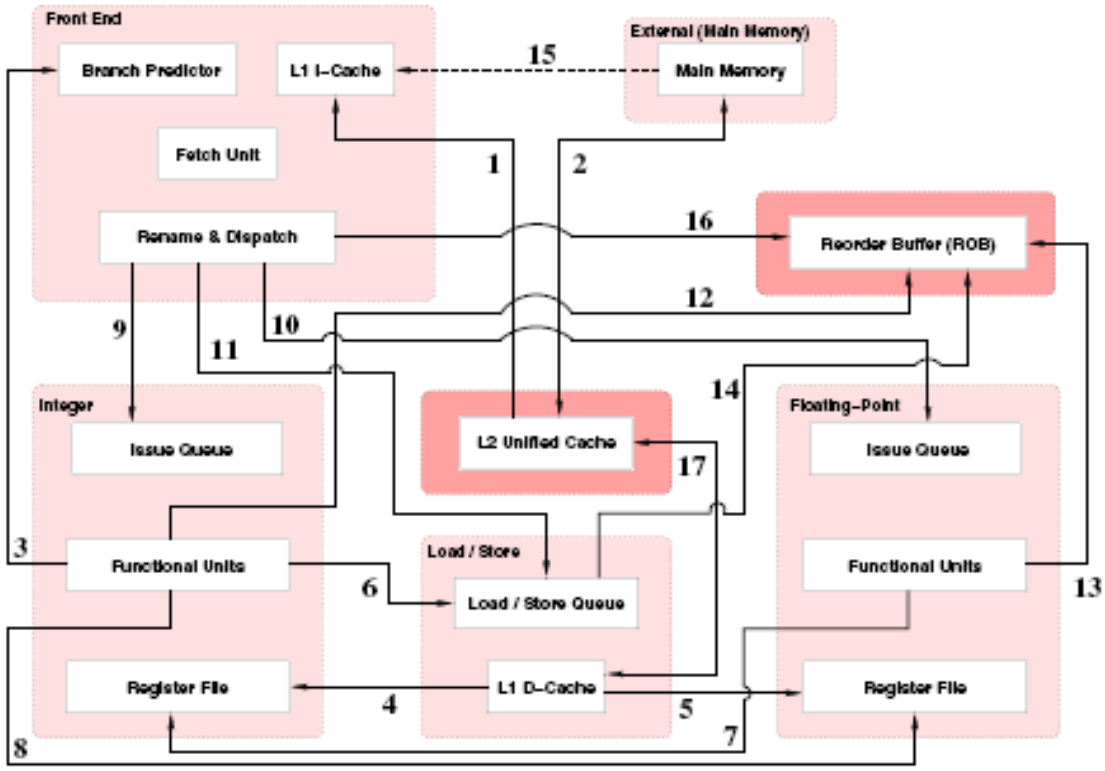


annote: bits de anotación
opcode: código de operación
target: destino del salto

3.3) Microarquitectura

La arquitectura que estudiamos se basa en la microarquitectura GALS propuesta por David H. Albonesi y su equipo [7, 27]. Es una microarquitectura formada por 5

dominios de reloj (la memoria principal está en un dominio diferente, pero no la contamos como dominio).



Cada dominio tiene su propio reloj local no correlacionado con los de los demás dominios. Por tanto, al comunicarse entre sí dos de ellos es necesario incluir un mecanismo de sincronización. En esta sincronización se puede perder hasta un ciclo de reloj, lo cual denominaremos penalización en el canal de comunicación. Estas penalizaciones pueden influir negativamente en el rendimiento del procesador, sobre todo si en alguna de ellas hay un cuello de botella. Los dominios de nuestra microarquitectura son las siguientes:

- Dominio del front-end
- Dominio del ROB
- Dominio del segundo nivel de caché
- Dominios de enteros y load/store
- Dominio de punto flotante

El dominio front-end está formado por el predictor de saltos, el primer nivel de caché de instrucciones, la unidad de fetch y las unidades de dispatch y renombramiento de registros.

El ROB está separado en un único dominio. Forma parte del camino crítico de cualquier instrucción (etapa commit y dispatch). Es por esto que se separa en un nuevo dominio trabajando siempre a la máxima frecuencia.

Debido a su tamaño si el segundo nivel de caché trabajara a una frecuencia relativamente alta consume mucha potencia. Por ello se encuentra en un dominio de reloj separado trabajando siempre a frecuencia media (si estuviera en el mismo dominio que el primer nivel de caché, el segundo nivel se vería obligado a trabajar a una frecuencia alta lo que ocasionaría una degradación en el rendimiento).

El dominio de punto-flotante está formado por la cola de lanzamiento de instrucciones en punto flotante, por las unidades funcionales y el archivo de registros físicos.

Aquellos canales que implican acceso a la caché son los que más influyen negativamente en el rendimiento. Es por esto que se mezclan los dominios de enteros y load/store en un único dominio.

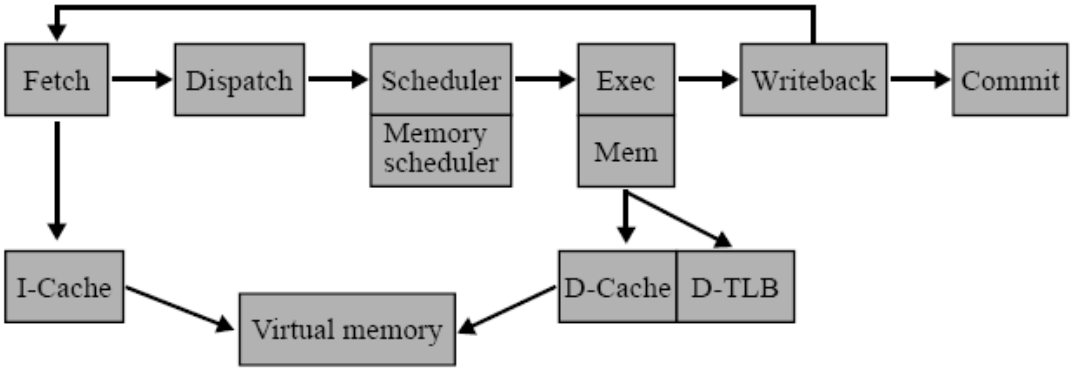
Todos los dominios de reloj trabajan a la misma frecuencia igual a la del circuito síncrono equivalente.

Descripción de los canales:

Canal 1	Cache L2=>Cache IL1
Canal 2	Memoria Principal=>Cache L2
Canal 3	ALU Enteros=>Predictor de saltos
Canal 4	Cache DL1=>Registros Enteros
Canal 5	Cache DL1=> Registros FP
Canal 6	Bus Resultados enteros => Cola Load/Store
Canal 7	Bus Resultados FP => Cola Load/Store
Canal 8	Bus Resultados Enteros => Registros FP
Canal 9	Cola Fetch => Cola Issue Enteros
Canal 10	Cola Fetch => Cola Issue FP
Canal 11	Cola Fetch => Cola Load/Store
Canal 12	ALU Enteros => ROB
Canal 13	ALU FP => ROB
Canal 14	Cola Load/Store => ROB
Canal 15	Memoria Principal =>Cache IL1
Canal 16	Cache L2 => Cache DL1

Etapas:

Se muestran a continuación las diferentes etapas del pipeline explicándolas. El pipeline utilizado en nuestra simulación es aquel que aparece en [26].

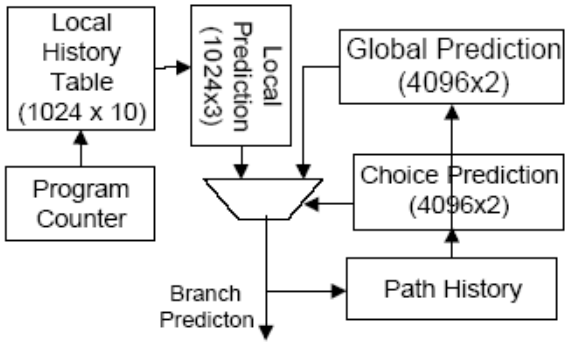


Etapa de fetch:

En la etapa de fetch se hace la búsqueda de instrucciones. Para seleccionar las instrucciones utilizamos la política ICOUNT2.8, es decir, se lanzan hasta un máximo de 8 instrucciones de cada hilo. Esta política da más prioridad a aquellos hilos que tienen menos instrucciones repartidas en las etapas de decodificación, renombramiento y en las colas de lanzamiento. Estos hilos son los que tienen menor probabilidad de bloquear la cola de lanzamiento, es posible que adelanten más trabajo, y proporcionan un mayor paralelismo.

En esta etapa se accede a la caché L1 de instrucciones para buscar las que se van a ejecutar. Es asociativa de dos vías de 16 KBytes. En caso de que las instrucciones no se encuentren en el primer nivel, se buscan en el segundo, que es unificado (de datos e instrucciones). El segundo nivel es directo (de una vía) de 256 Kbytes. Una vez leídas las instrucciones se añaden a la cola de dispatch.

Además de leer las instrucciones, en esta etapa se realiza la predicción de saltos mediante un predictor híbrido que elige dinámicamente entre un predictor de historia local y otro de historia global. Ambas tablas guardarán la información de los saltos de ambos hilos, por lo que en cada entrada debe incluir un identificador de hilo (tag) para saber a qué hilo hace referencia la información.



La tabla de historia local contiene los últimos diez comportamientos de 1024 saltos indexados (1024 entradas) por los diez bits menos significativos del contador de programa (hay un contador de programa por cada hilo). A partir de estos diez bits, se

extrae la información necesaria para decidir si el salto se toma o no (usando un contador saturado), la cual se guarda en la tabla de predicción local.

La tabla de predicción global es una tabla con 4096 entradas formada por contadores saturados de dos bits, indexada por el camino global de los últimos 12 saltos.

Se usa un juez (contador saturado) para decidir si se debe escoger la predicción local o la global. Ambas predicciones se obtienen a partir del bit más significativo de la tabla de predicción local y de la tabla de predicción global. Una vez resuelto el salto (en la etapa de Writeback) se actualizan tanto las tablas como el juez.

Etapa de dispatch:

En esta etapa se realiza la decodificación de las instrucciones y el renombramiento de registros. El renombramiento se realiza con el fin de evitar riesgos de datos de tipo EDE y EDL

Los riesgos que hay son:

- Escritura después de escritura (EDE)
- Escritura después de lectura (EDL)
- Lectura después de escritura (LDE)

EDE: Se da cuando dos instrucciones que escriben sobre un mismo registro, lo hacen en desorden (escribe primero aquella que debería escribir después). Tras la segunda escritura el registro está actualizado con un dato erróneo (el que debía tener antes de la última escritura que ya se hizo).

EDL: Se da cuando una lectura lee un dato que ha sido escrito previamente, y que debería ser escrito después.

LDE: Se da cuando una lectura lee un dato que debía haber sido escrito previamente pero no ha sido así.

Esta arquitectura, tiene ejecución fuera de orden, por lo que pueden escribir fuera de orden y por tanto aparecen los riesgos EDE y EDL.

Estos riesgos son fácilmente evitados usando el renombramiento de registros, que asocia a cada instrucción que va a producir un resultado que alojará en un registro uno nuevo.

Las dependencias LDE se mantienen y son resueltas en la siguiente etapa (etapa de Scheduler).

Usamos 200 registros de punto flotante y 200 de enteros por cada hilo. Al final de esta etapa las instrucciones se guardan en las colas de enteros, de punto flotante y de load/store dependiendo del tipo de instrucción. También se guardan en el reorder buffer (ROB).

Etapa de Scheduler:

En esta etapa las instrucciones que están en las diferentes colas y que tienen sus operandos disponibles, se van lanzando a las unidades funcionales adecuadas, o a la memoria. Cuando un resultado de una unidad funcional o de un load está disponible se notifica a aquellas instrucciones que están en las colas esperando por este resultado. En cada ciclo se pueden lanzar un máximo de 11 instrucciones.

Etapa de exec:

En la etapa de ejecución las instrucciones de load acceden a la caché L1 de datos, para extraer el dato que se necesita. En caso de no encontrarse el dato en la caché L1, accede a la caché L2 que es unificada como hemos mencionado anteriormente. La caché L1 de datos es una caché directa y tiene un tamaño de 32 KBytes.

Tenemos dos sumadores dedicados para calcular la dirección efectiva, 8 sumadores de enteros y 4 de punto flotante. Dos multiplicadores, uno de enteros y otro de punto flotante.

Etapa writeback:

En esta etapa se detectan aquellas instrucciones que han finalizado. Notifica la finalización de la instrucción, de forma que las dependientes que necesiten el dato generado, puedan tomarlo.

En esta etapa también se detecta si el salto se tomó correctamente o no. En caso de que estuviese mal predicho, se descartan las instrucciones lanzadas erróneamente. Y en cualquier caso se actualizan las tablas de predicción y el juez.

Etapa commit:

En esta etapa se hace (en el orden del programa) el "commit" de aquellas instrucciones que ya se han ejecutado (posiblemente fuera de orden). Cuando una instrucción finaliza, su resultado se escribe en el archivo de registros físicos. Se puede hacer commit de hasta 24 instrucciones (de las últimas 24 que hayan finalizado) de forma independiente para cada hilo.

4) Simulación

4.1) Introducción

El procesador que estudiamos es un procesador SMT-GALS. Debido a estas características, los diferentes dominios de reloj se comunican entre sí por medio de canales asíncronos. Esto hace que aparezcan penalizaciones debidas a la sincronización entre los diferentes módulos.

Nuestro objetivo es estudiar el comportamiento de diferentes canales y ver cuáles de ellos influyen más en el rendimiento total del procesador, así como encontrar posibles soluciones para un mejor funcionamiento. También esperamos que el hecho de que la arquitectura tenga características SMT, haga que la ejecución de dos hilos simultáneamente proporcione una mejora significativa sobre una ejecución monohilo.

Realizaremos diferentes simulaciones de un conjunto de benchmarks (SPEC2000), comparando los resultados para diferentes tipos de simulación (monohilo y SMT).

También estudiaremos los parámetros de la arquitectura que hacen que el comportamiento sea óptimo sin necesidad de tener unos recursos elevados.

4.2) Pruebas para el ajuste de la arquitectura

Dado que la arquitectura inicial que se nos facilitó no se había probado con anterioridad, sus parámetros no eran los adecuados para aprovechar las ventajas que ofrece el uso simultáneo de más de un hilo de ejecución (SMT). Por ello, lo primero que haremos será ajustar los parámetros para obtener un buen rendimiento SMT.

Una vez tengamos la nueva arquitectura ajustada, realizaremos el estudio de los canales, viendo cuáles afectan más al rendimiento, con el fin de proponer cambios en los dominios de reloj de la arquitectura.

En la siguiente tabla se muestran los parámetros iniciales de nuestra arquitectura:

Parámetros de configuración	**Valor**
Predictor de saltos :	
Nivel 1	1024
Nivel 2	1024
Predictor bimodal	2048
Predictor combinado	4096
BTB	4096, 2 vías
Penalización predicción errónea	9 ciclos
Anchura de Decode	8
Anchura de Issue	6
Anchura de Commit	11
Cache L1D	32KB, 1 vía
Cache L1I	16KB, 2 vías
Cache L2	256KB, 1 vía
Latencia cache L1	2 ciclos
Latencia cache L2	12 ciclos
ALU enteros	4
ALU FP	2
Cola Issue Enteros	32 entradas
Cola Issue FP	32 entradas
Cola Load/store	64 entradas
Registros físicos/hilo	48 FP, 48 Enteros
ROB/hebra	128

Estos datos con los que comenzamos, son fruto de la integración realizada por Sonia López Alarcón (DACYA) del diseño SMT de Ali El Moursy y el GALS de David H. Albonesi. A continuación se realizan las primeras pruebas de ajuste de la microarquitectura.

.

Las simulaciones se han realizado sobre diferentes benchmarks, todos ellos forman parte de los SPEC2000, y hemos utilizado una ventana de simulación de 100 millones de instrucciones (200 millones de instrucciones en la ejecución de dos hilos).

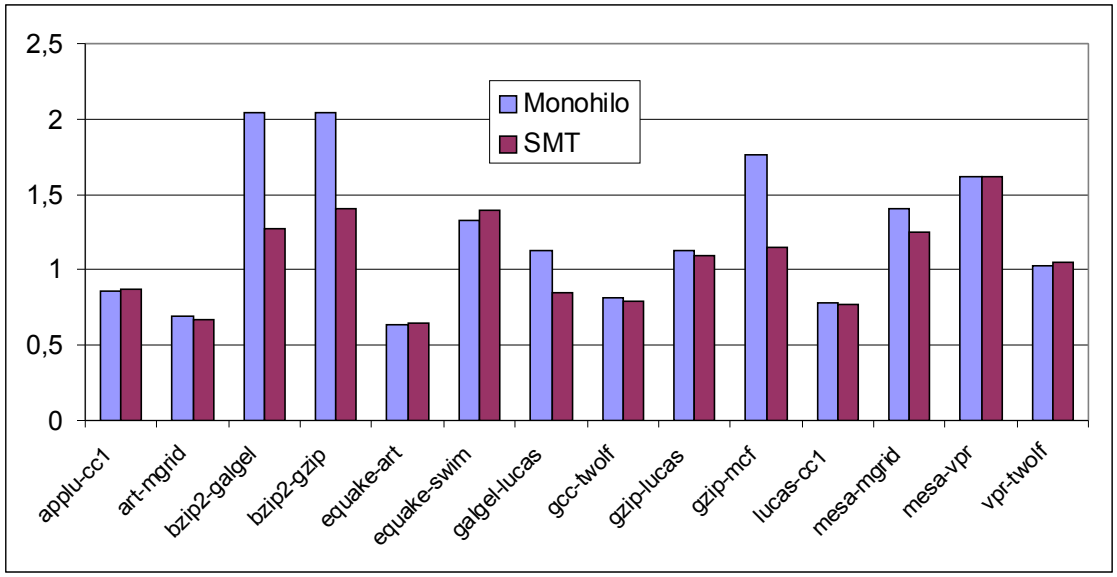
Realizamos dos tipos de simulaciones diferentes para compararlas entre sí:
- Simulación monohilo: Simulamos la ejecución de un único benchmark.

- Simulación SMT: Simulamos la ejecución de un par de benchmarks simultáneamente (dos hilos).

Para la sintonización SMT comprobaremos si la arquitectura base cumple las expectativas de una arquitectura de este tipo. Para ello analizaremos la mejora obtenida en la ejecución SMT frente a la ejecución monohilo. Lo haremos comparando sus IPC's. Si no tiene el comportamiento esperado, modificaremos los parámetros de la arquitectura que fueran necesarios.

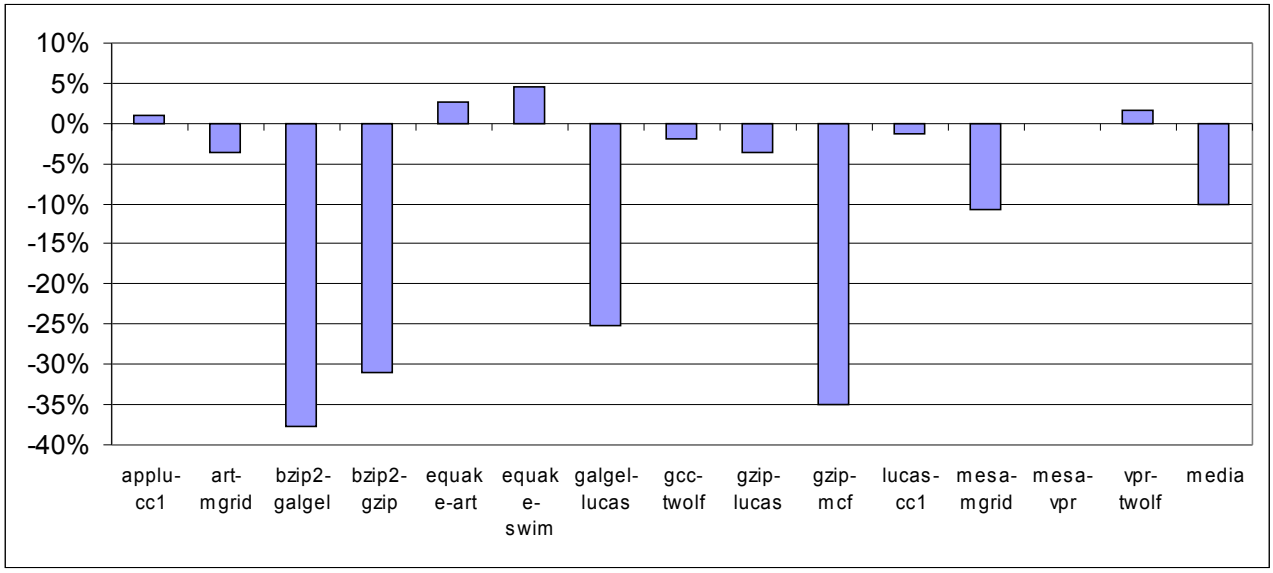
En la ejecución SMT obtenemos el IPC directamente de la simulación. Para la ejecución monohilo obtenemos los CPI's de las ejecuciones de cada uno de los hilos, y calculamos el IPC medio, como la inversa de la media de los CPI's.

La siguiente gráfica muestra la comparativa entre una ejecución monohilo y una SMT:



Como se puede ver en la gráfica, en casi todos los casos, el rendimiento disminuye en la ejecución con dos hilos, resultado que no era el esperado.

En la siguiente gráfica se muestra la relación entre el tiempo de ejecución de dos hilos simultáneamente (con SMT) respecto al de los mismos hilos ejecutados secuencialmente (es decir, el speedup de la ejecución SMT frente a la monohilo). Vemos que esta razón de mejora es negativa: empeora.



Viendo que en media se reduce el rendimiento en un 10% en la ejecución con dos hilos (que debería mejorar por el mayor paralelismo de un SMT), llegamos a la conclusión de que tal vez la competencia por los recursos entre los dos hilos era muy grande. La arquitectura que usamos para estas simulaciones tenía unos recursos mínimos, por lo que cabía la posibilidad de que ambos hilos estuvieran mucho tiempo compitiendo entre sí, lo que implicaba una pérdida de rendimiento.

Al ver que los resultados no fueron los esperados, decidimos ampliar los recursos de la arquitectura para intentar mejorar el rendimiento. Para esto estudiamos las posibles causas de la degradación del rendimiento obtenida al ejecutar dos hilos simultáneamente.

Los resultados obtenidos no son fruto de la primera modificación que hicimos en nuestra arquitectura, si no que hicimos varias pruebas con distintos parámetros de la arquitectura hasta obtener la versión final.

Cambiamos muchos parámetros respecto a la configuración de la arquitectura inicial.

Aumentamos el número de instrucciones que pueden hacer el commit hasta 24, para que el ROB no esté lleno innecesariamente. También aumentamos el tamaño del ROB, que pasó a tener de 256 entradas a 512 en una primera versión modificada.

Nos dimos cuenta de que el ROB tenía la ocupación muy baja, por lo que estaba claro que no alimentábamos al pipeline con instrucciones de forma adecuada. Si al ROB no le llegan suficientes instrucciones, significa que en alguna etapa se atascaban y así la microarquitectura siempre trabajaba por debajo de sus posibilidades.

Para solucionar esto, probamos con diferentes tamaños de cola de fetch, pero finalmente nos dimos cuenta de que el problema estaba en el número de registros que usábamos. Antes teníamos 96 registros de enteros y punto flotante. En nuestra última versión de la arquitectura usamos 400 de enteros y punto flotante (200 para cada hilo).

Finalmente vimos que el ROB estaba actuando por debajo de sus posibilidades y que un tamaño tan grande no era conveniente, por lo que lo rebajamos al tamaño original, 256 entradas (128 para cada hilo).

También usamos dos sumadores dedicados, y aumentamos el número de sumadores de enteros de 4 a 8, y los de punto flotante de 2 a 4.

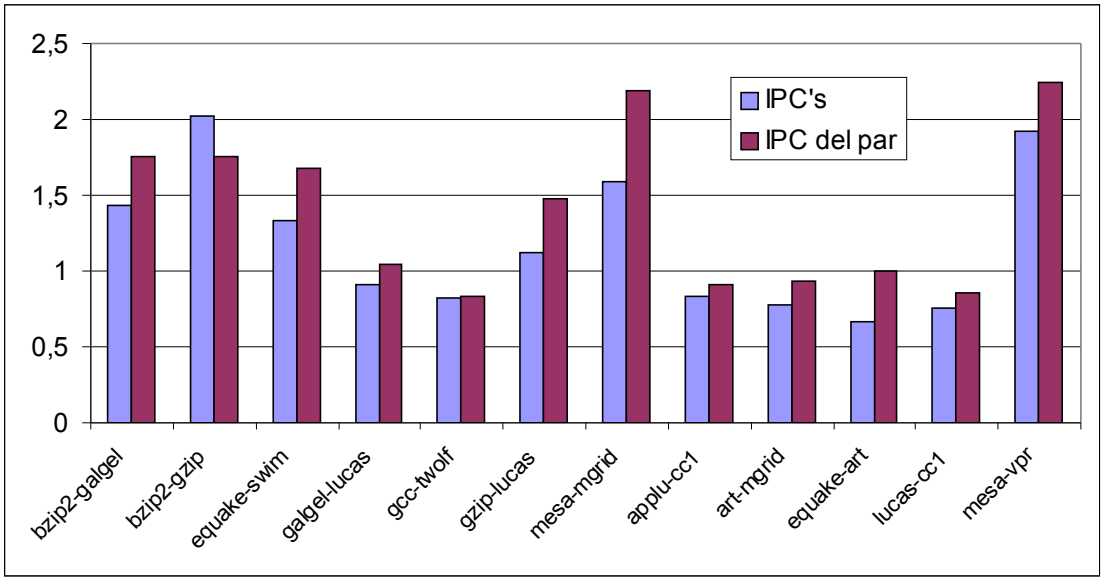
A continuación mostramos una tabla con el resumen de los datos de nuestra nueva arquitectura, subrayando en negrita aquellos cambios que introdujimos en la configuración de la arquitectura inicial.

Parámetros de configuración	**Valor**
Predictor de saltos :	
Nivel 1	1024 entradas
Nivel 2	**4096 entradas**
Predictor bimodal	2048
Predictor combinado	4096
BTB	4096, 2 vías
Penalización predicción errónea	9 ciclos
Anchura de Decode	8
Anchura de Issue	**11**
Anchura de Commit	**24**
Cache L1D	32KB, 1 vía
Cache L1I	16KB, 2 vías
Cache L2	256KB, 1 vía
Latencia cache L1	2 ciclos
Latencia cache L2	12 ciclos
ALU enteros	**8**
ALU FP	**4**
Cola Issue Enteros	32 entradas
Cola Issue FP	32 entradas
Cola Load/store	**256 entradas**
Registros físicos/hilo	**200 FP, 200 Enteros**
ROB/hebra	128

A continuación, una vez ajustados los parámetros de la arquitectura, probamos que efectivamente hemos conseguido mejorar el rendimiento de con una ejecución multihilo (SMT) frente a la ejecución monohilo.

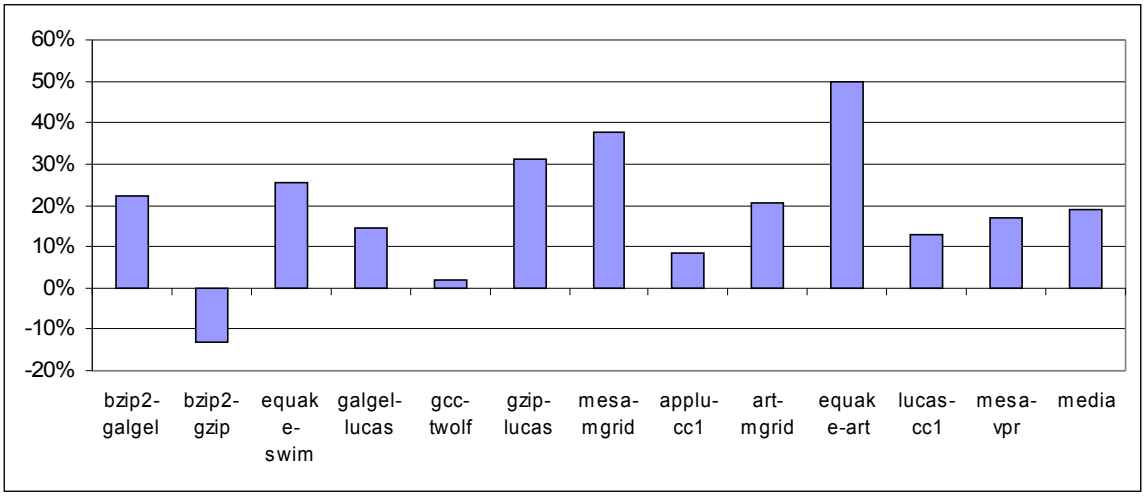
En la siguiente gráfica mostramos el IPC obtenido tanto para una simulación monohilo como una SMT. Como se puede ver, el IPC mejora en todos los pares estudiados excepto en uno. Este par corresponde a las aplicaciones bzip2-gzip, ambas

son de compresión de datos, y al competir ambos hilos por los mismos recursos, el rendimiento empeora.



Descontando el caso del par (bzip2-gzip), los resultados son los esperados. El objetivo era que cuando un hilo no estuviera haciendo trabajo útil debido quizás a un fallo de caché o por las dependencias, el otro hilo pudiera aprovechar la situación para adelantar trabajo y utilizar los recursos de forma eficiente, y se puede observar en las gráficas cómo aumenta en la ejecución con dos hilos.

La siguiente gráfica muestra la mejora relativa de IPC's (speedup) de la simulación SMT respecto a la monohilo.



La mejora media está un poco por debajo del 20 %, a pesar de que en el caso bzip2-gzip el rendimiento disminuya al ejecutar el par de benchmarks.

Así pues, hemos conseguido una arquitectura con parámetros que explotan el diseño SMT. Ahora, trabajando sobre esta arquitectura buscaremos qué canales de

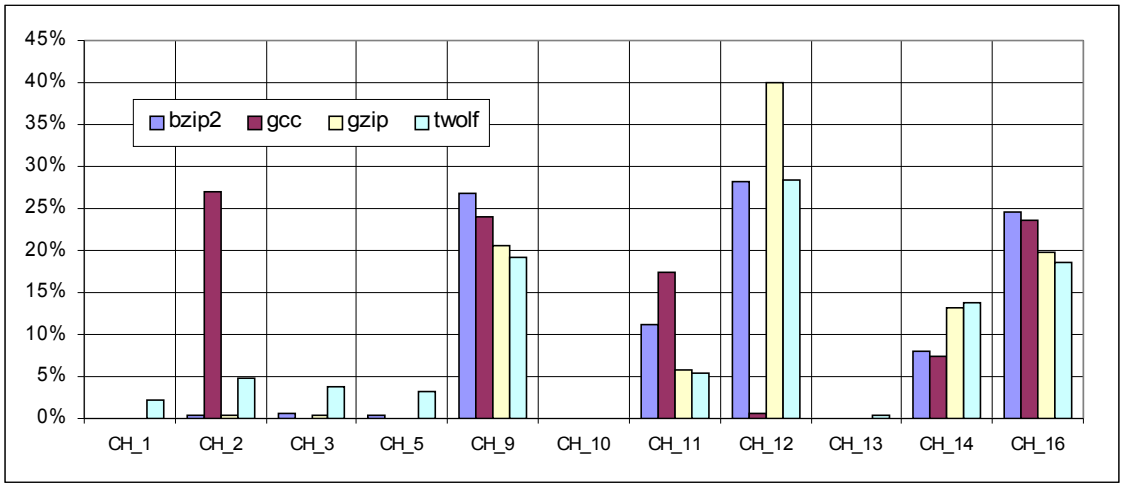
comunicación tienen más penalizaciones y cuáles afectan más al rendimiento, con el objetivo de proponer la división en los dominios de reloj óptimos.

4.3) Estudio de las penalizaciones

Al ser un diseño con múltiples dominios de reloj, el tiempo de ejecución incluye las penalizaciones debidas a la sincronización entre los diferentes dominios.

A continuación se muestran dos gráficas que representan el porcentaje de penalizaciones ocurridas en cada canal respecto al total de penalizaciones de la simulación.

En la primera se puede ver el porcentaje de penalizaciones de 4 benchmarks enteros en simulación monohilo.

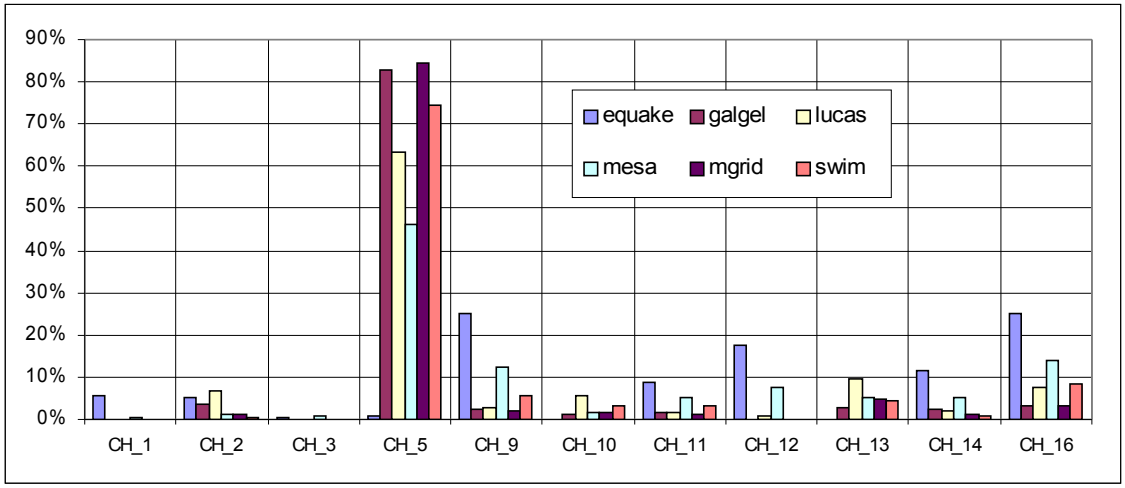


Observamos que los canales 12, 9 y 16 son los que más penalizaciones generan.

El canal 9 conecta la etapa de dispatch y renombramiento con la cola de lanzamiento de enteros. Al ser una gráfica que representa benchmarks de enteros, esto tiene sentido, ya que la mayoría de las instrucciones serán enteras y tendrán que recorrer este camino.

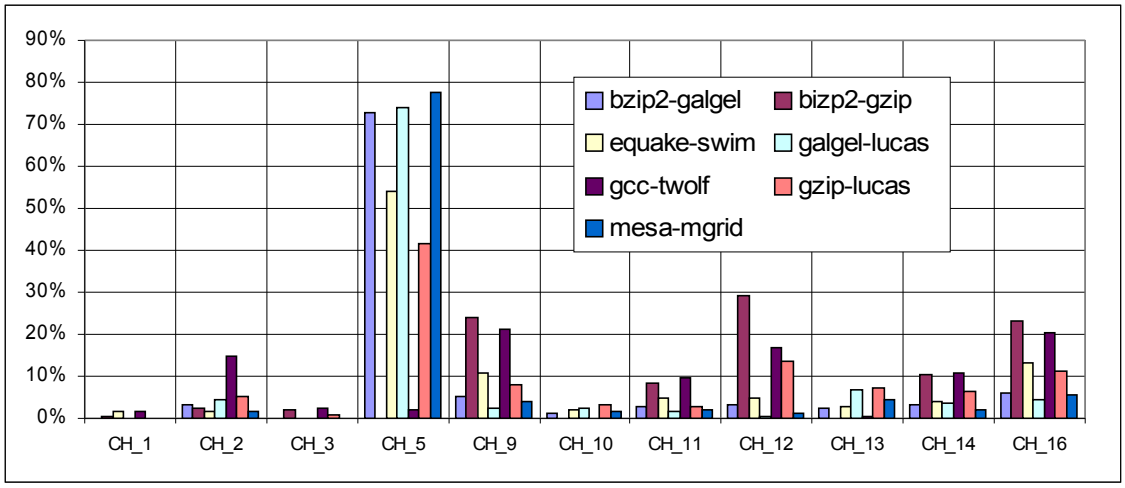
Algo similar ocurre con los canales 12 y 16, ambos unen el dominio del ROB con unidad de ejecución de los enteros (canal 12) y con la etapa de dispatch (canal 16). Estos resultados también son esperados ya que el ROB entra en juego con cada nueva instrucción que se ejecuta en el pipeline (tanto al inicio como al final), y con cada nuevo resultado que se genera.

En la siguiente gráfica representamos la misma información (porcentaje de penalizaciones) para 6 benchmarks de punto flotante en una simulación de tipo monohilo.



Ahora cabe resaltar las penalizaciones en el canal 5, canal que comunica los registros de punto flotante con el primer nivel de la caché de datos. Las penalizaciones debidas a este canal llegan a un máximo que ronda el 80 por ciento de las penalizaciones totales, mientras que en el caso anterior, el mayor porcentaje de penalización que se conseguía para algún canal (el 12 en particular), llega a un 40 por ciento.

La siguiente gráfica muestra el porcentaje de penalizaciones en cada canal en la simulación SMT.



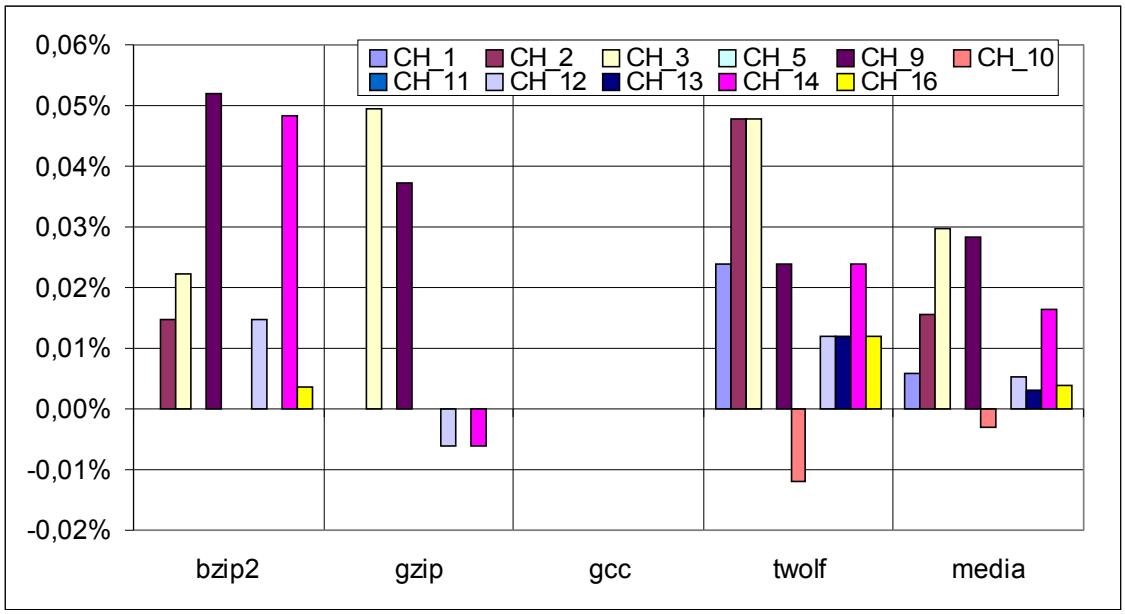
Se puede ver cómo, aunque aquellos pares de enteros (bzip2-gzip) no presentan penalizaciones en el canal 5, este canal sigue siendo el dominante en cuanto a penalizaciones. Por ejemplo, bzip2-galgel (siendo de enteros y punto flotante respectivamente), presenta en el canal cinco más de un 70 por ciento de las penalizaciones (debidas únicamente al galgel).

Hay que destacar que el porcentaje de penalizaciones no refleja el impacto en el rendimiento. Las penalizaciones pueden solaparse debido a fallos de caché o a la resolución de alguna dependencia, lo que haría que el rendimiento no sufriera un impacto debido a esta penalización.

También en algunos casos son las propias penalizaciones las que se apantallan unas a otras. Para identificar el impacto en el rendimiento de la penalización de cada canal, hemos simulado cada aplicación (tanto monohilo como SMT) eliminando las penalizaciones en cada canal, y comparando los resultados obtenidos con los resultados de la simulación sin eliminarlas.

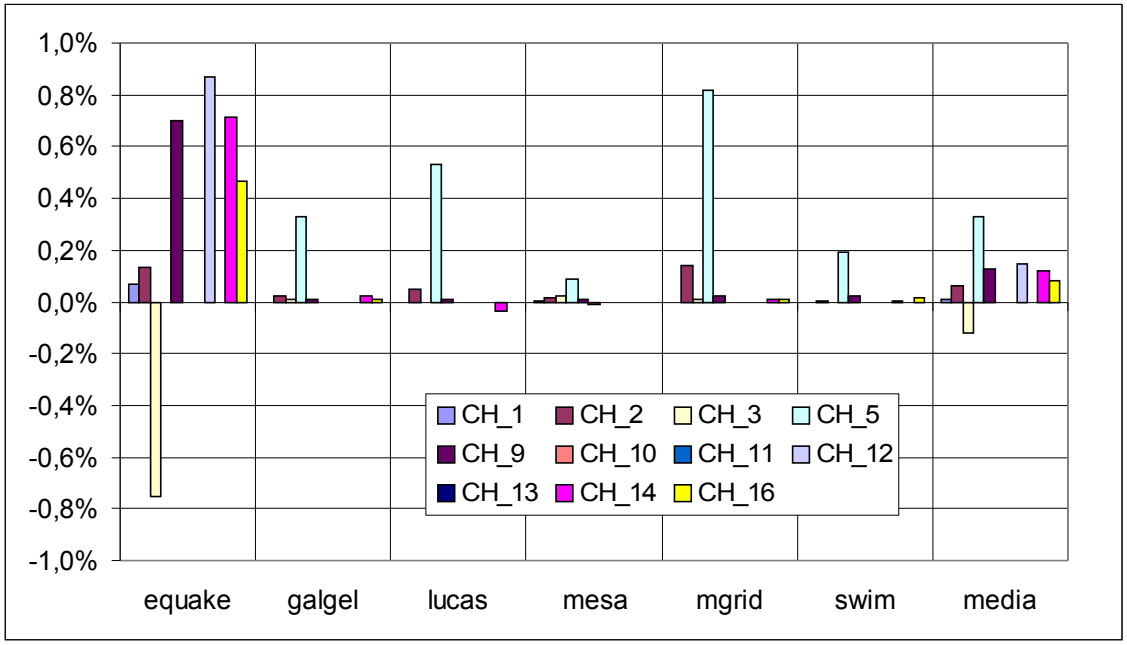
Las siguientes gráficas muestran el impacto en el rendimiento cuando se eliminan las penalizaciones en cada canal (CH_X). Este impacto está calculado con respecto al diseño MCD con todas las penalizaciones.

La siguiente gráfica está basada en los resultados para benchmarks de enteros.



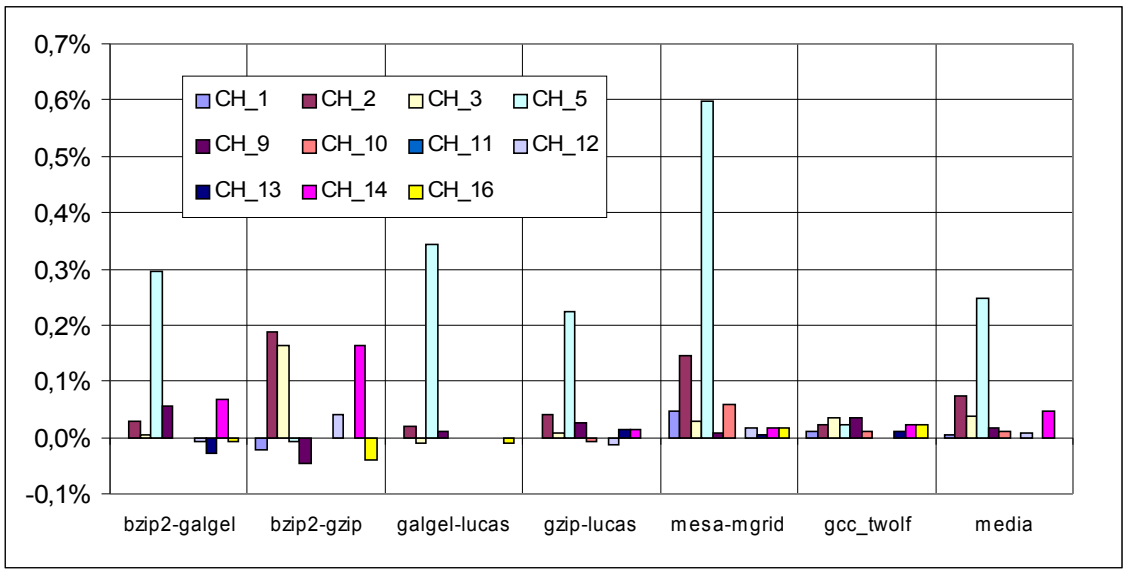
Aquí se puede observar que el impacto es mínimo. Lo máximo que aumenta el rendimiento es del orden de 0,05 %.

La siguiente gráfica representa la misma información para benchmarks de aplicaciones en punto flotante.



Para las aplicaciones en punto flotante, como ya se observaba en las gráficas anteriores, es el canal 5 aquel que tiene un mayor impacto en el rendimiento. Además, el máximo aumento de rendimiento que se consigue al quitar las penalizaciones debido a este canal ronda el 0,9 %, significativamente más grande que el máximo aumento de rendimiento que se conseguía en la gráfica anterior para las aplicaciones de enteros y que estaba en torno al 0,05%.

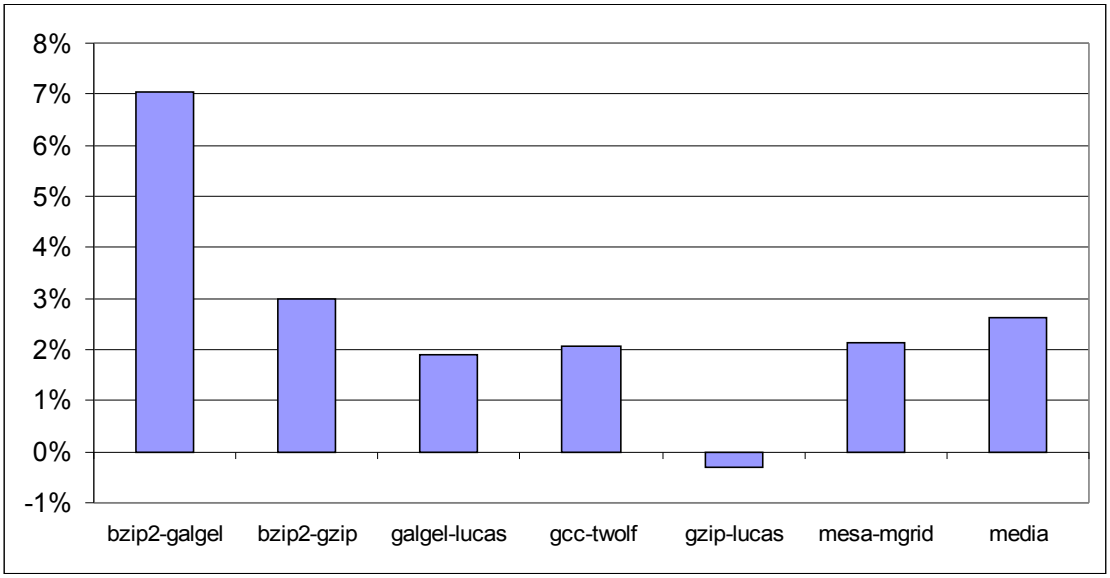
A continuación se muestra el mismo tipo de gráfica pero para simulación con SMT:



Al simular los pares, este impacto se reduce un poco respecto a aquel en las aplicaciones de punto flotante, pero aún así sigue siendo el canal 5 el responsable de causar las mayores penalizaciones.

Impacto en el rendimiento:

En la siguiente gráfica se puede ver el empeoramiento relativo del IPC expresada en tanto por ciento, de la arquitectura que estudiamos con múltiples dominios de reloj, respecto al diseño síncrono. La degradación de rendimiento que producen estos canales no supera el 3% en media. Sin embargo, este empeoramiento se ve compensado por un diseño más sencillo debido a la ausencia de un reloj global y los problemas que ello conlleva.



4.4) Conclusiones y trabajo futuro:

El diseño SMT produce una mejora en el rendimiento. La ejecución simultánea con dos hilos hace que los recursos se utilicen más eficientemente y explota el paralelismo tanto a nivel de hilo como a nivel de instrucción. La mejora que aporta la ejecución de dos hilos, está en media en torno al 20% con respecto a la ejecución de un solo hilo.

El diseño GALS, por otra parte, aporta cierta degradación en el rendimiento. Sin embargo, ésta no es muy significativa (haciendo una comparativa entre 6 pares de benchmarks, la pérdida de rendimiento está, en media en torno a un 3%), y usándolo junto con las características SMT, hemos demostrado que se consiguen mejoras en la ejecución de dos hilos respecto a la de un único hilo.

Nuestros resultados dejan claro que es la comunicación entre la caché de datos de primer nivel y los registros de punto flotante (canal 5) la que causa un mayor impacto en cuanto a rendimiento.

Lo mismo ocurría en el diseño MCD anterior a las modificaciones de Albonesi, entre los registros de enteros y la caché de datos de primer nivel. Su solución fue unir el dominio de enteros con el de Load/Store. Esto hace pensar que la posible solución al

problema análogo pero con los registros de punto flotante, sea juntar también los dominios (de Load/Store y punto flotante), lo cual sólo sería razonable para aquellas aplicaciones que hacen amplio uso de instrucciones en punto flotante.

Otro de los canales que más afectan al rendimiento es el 9, aquél que conecta el dominio front-end con el dominio de los enteros. Afecta mucho más para aquellas aplicaciones enteras que para las de punto flotante, como era de esperar. Se podría realizar la unión de estos dos dominios en uno sólo. Habría que estudiar las consecuencias de este cambio, y si la mejora es mayor que aquella ya realizada en la que el dominio de los enteros se mezcla con el de Load/Store.

El canal 2 (aquél que conecta la memoria principal con el segundo nivel de caché) también afecta en el rendimiento total en cuanto a penalizaciones. Las penalizaciones en este canal se deben únicamente a fallos de caché, por tanto se podría mejorar el segundo nivel de caché, reduciendo así el número de accesos a memoria principal.

Como trabajo futuro, quedaría probar las diferentes propuestas mencionadas anteriormente y estudiar su conveniencia. También se ha de realizar el mismo estudio para la ejecución de más de dos hilos, modificando para ello las características de la microarquitectura correspondientes.

Este estudio ha sido realizado con frecuencias de reloj fijas para cada dominio. Queda pendiente estudiar el comportamiento de la microarquitectura usando técnicas que permitan a los diferentes dominios adaptar dinámicamente los valores de frecuencia/voltaje o frecuencia/recursos, en función de las necesidades de la aplicación.

5) BIBLIOGRAFÍA:

[1] Matthew Heath y Ian Harris, "A Deterministic Globally Asynchronous Locally Synchronous Microprocessor Architecture", en Proceedings of the Fourth International Workshop on Microprocessor Test and Verification Common Challenges and Solutions (MTV'03), IEEE 2003

[2] Besad Mesgarzadeh y Christer Zvensson, "A new mesochronous clocking scheme for sinchronization in SoC", en ISCA 2004

[3] Tony Werner y Venkatesh Akella, "Asynchronous Processor Survey", en IEEE Noviembre 1997

[4] E. Brunvand, "The NSR Processor", en IEEE 1993

[5] Matthew W. Heath, Member, IEEE y Wayne P. Burleson, Senior Member, IEEE, "Synchro-Tokens: A Deterministic GALS Methodology for Chip-Level Debug and Test", en IEEE TRANSACTIONS ON COMPUTERS, VOL. 54, NO. 12, DECEMBER 2005

[6] Simon Moore y George Taylor, "Point to Point GALS Interconnect", en Proceedings of the Eighth International Symposium on Asynchronous Circuits and Systems (ASYNC.02), IEEE 2002

[7] Anoop Iyer y Diana Marculescu, "Power and Performance Evaluation of Globally Asynchronous Locally Synchronous Processors", en Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA.02)

[8] Steven Dropsho, Greg Semeraro, "Dynamically Trading Frequency for Complexity in a GALS Microprocessor", en Proceedings of the 37th International Symposium on Microarchitecture (MICRO-37'04), IEEE 2004

[9] YongKang Zhu y David H. Albonesi, "A High Performance, Energy Efficient GALS ProcessorMicroarchitecture with Reduced Implementation Complexity", en IEEE 2005

[10] Ali El-Moursy y David H. Albonesi, "Front-End Policies for Improved Issue Efficiency in SMT Processors", en Proceedings of the 9th International Conference on High Performance Computer Architecture

[11] L.A. Plana y P.A. Riocreux, "SPA - A Synthesisable Amulet Core for Smartcard Applications", en Proceedings of the Eighth International Symposium on Asynchronous Circuits and Systems (ASYNC.02), IEEE 2002

[12] Aristides Efthymiou y Jim D. Garside, "Adaptive Pipeline Structures for Speculation Control", en Proceedings of the Ninth International Symposium on Asynchronous Circuits and Systems (ASYNC'03), IEEE 2003

[13] Greg Semeraro y David H. Albonesi, "Hiding Synchronization Delays in a GALS Processor Microarchitecture", en Proceedings of the International Symposium on Asynchronous Circuits and Systems, ASYNC'04

[14] Grigorios Magklis y Michael L. Scout, “Profile-based Dynamic Voltage and Frequency Scaling for a Multiple Clock Domain Microprocessor”, en ISCA’03

[15] Rostislav (Reuven) Dobkin1 y Ran Ginosar, “Data Synchronization Issues in GALS SoCs”, en Proceedings of the 10th International Symposium on Asynchronous Circuits and Systems (ASYNC’04), IEEE 2004

[16] D.M. Tullsen, S.J. Eggers, and H.M. Levy. “Simultaneous multithreading: Maximizing on-chip parallelism”. 22nd Annual International Symposium on Computer Architecture,June 1995.

[17] S. Egger, J. Emer, H. Levy, J.Lo, R. Stamm and D. Tullsen, “Simultaneous Multithreading: A Platform for Next-Generation Processors,” IEEE Micro, 1997.

[18] S. Raasch and S. Reinhardt, "Applications of Thread Prioritization in SMT Processors", Proc. of Multithreaded Execution, Architecture, and Compilation Workshop (MTEAC), Febrero 1999.

[19] T. Wang, F. Blagojevic y D. S. Nikilopoulos. “Runtime Support for Integrating Precomputation and Thread-Level Parallelism on Simultaneous multithreading Processors”. Proceedings of the 7th workshop on workshop languages, compilers and run-time support for scalable systemes. 2004

[20] A. Roth y G. Sohi. “Speculative Data-Driven Multithreading”. In Proc. of the 7th International Symposium on High-Performance Computer Architecture, Enero 2001

[21] K. Wilcox and S. Manne. “Alpha Processors: A history of power issues and a look to the future”. Cool Chips Tutorial, in conjunction with Micro-32, 1999.

[22] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo and R. Stamm. “Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor,” 23nd

[23] D.M. Tullsen. “Simulation and modeling of a simultaneous multithreading processor”. 22nd Annual Computer Measurement Group Conference, diciembre 1996

[24] A. El-Mourse y D.H_ Albonesi. “Front-end policies for improved issue efficiency in SMT processors”. Proceedings 9th. Conference on High Performance Computer Arquitecture, Febrero 2003.

[25] Miquel Pericas. “Simultaneous Multithreading: Present Developments and Future Directions”

[26] D. Burger and T. Austin. “The SimpleScalar Tool Set, Version2.0”. Technical Report CS-TR-97-1342, University of Wisconsin-Madison, Junio 1997.

[27] S. Dropsho, G. Semeraro, D.H. Albonesi, G.Magklis, yMichael L. Scott. “Dynamically Trading Frequency for Complexity in a GALS Microprocessor”.

Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture, 2004

Apéndice A: Información de los benchmark SPEC2000

Nombre	Autores	Categoría general del benchmark	Lenguaje en el que está programado
gzip	Jean-Loup Gailly	Compresión de datos	ANSI C
vpr (versatile place and route)	Vaughn Betz	Direccionamiento y ubicación de circuitos FPGA	ANSI C
gcc	Richard Stallman y un gran equipo de ayudantes	Optimizar el compilador del lenguaje C	ANSI C
mcf	Dr. Andreas Loebel	Optimización combinatoria/ Programación de un vehículo para que aparque solo	ANSI C (se requiere la librería matemática *libm*)
crafty	Robert Hyatt	Vídeo juego (ajedrez)	ANSI C
parser	Danny Sleator	Procesamiento de lenguaje natural	ANSI C
eon	Meter Shirley, Ken Chiu, Kart Zimmerman and Steve Marshmer	Efectos producidos por distintas fuentes de luz. Visualización por computador	C++
perlbmk	Larry Wall	Lenguaje de programación; PERL	
gap	Martin Schoenert	Teoría de grupos / Intérprete	ANSI C
vortex	Peter R. Homan	Base de datos orientada a objetos	C
bzip2	Julian Seward	Compresión de datos	ANSI C
twolf	Bill Swartz	CAD (Computer Arded Design) de ubicación y ruteo	

Nombre	Autores	Categoría general del benchmark	Lenguaje en el que está programado
wupwise	Institute of applied computer science university of Wippertal	Físicas / Cronodinámica cuántica	Fortran77
swim	Paul N. Swrztrauber	Meteorología: Modelado de aguas poco profundas	Fortran77
mgrid	Eric Barszcz, Paul O. Frederickson	resultor de multi-grid en campos potenciales 3D	
applu	Sisira Weeratunga	Problemas de dinámica de fluidos y física computacional (ecuaciones diferenciales parciales parabólicas y elípticas)	
mesa	Brian E. Paul	Librería de gráficos 3D (OpenGL)	
galgel	Alexander Gelfgat	Análisis de inestabilidad oscilatoria	
art	Charles Roberson and Max Domeika	Reconocimiento de imágenes y redes de trabajo neuronales	ANSI C
equake	David R. O'Hallaron and Loukas F. Callivokas	Simulación de la propagación de una señal sísmica en una cuenca	ANSI C
facerec	Jan C. Vorbrüggen	Procesamiento de imágenes (rostros)	Fortran90
ammp	Robert W. Harrisson	Modelado de largos sistemas de moléculas normalmente asociadas con biología	C
lucas	Ernst Mayer	Teoría de números: Test de primalidad	Fortran90
fma3d	Samuel W. Key	Simulación de elementos físicos en choque	Fortran90
sixtrak	Frank Schmidt	Modelo de un acelerador de partículas de alta energía	Fortran77
apsi	Zaphiris D. Chritidis, IBM Yorktown Heights	Predicción del tiempo (resuelve problemas de temperatura, viento y distribución de contaminantes)	

Se autoriza a la Universidad Complutense a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la propia memoria, como el código, la documentación y/o el prototipo desarrollado.

Fdo:

Omar Aragón

César Boullosa Serrano

Tania Orell Orgaz

.