Implementando recuperaciones precisas en procesadores con consolidación fuera de orden

Isidro González[1], Oliverio J. Santana[2], Alex Pajuelo[1] y Mateo Valero[1,3]

[1] Dep. Arquitectura de Comp.
Univ. Pol. de Catalunya
{iglez,mpajuelo}@ac.upc.edu

[2] Dep. Informática y Sistemas
Univ. Las Palmas de G.C.
ojsantana@dis.ulpgc.es

[3] Barcelona Supercomputing Center
Centro Nacional de Supercomputación
mateo.valero@bsc.es

Resumen—**La creciente diferencia entre la velocidad del procesador y la memoria (*memory wall*) constituye una seria limitación para el rendimiento de los procesadores. Como solución a este problema se han propuesto arquitecturas con consolidación (*commit*) fuera de orden que permiten expandir virtualmente la ventana de instrucciones. Estos mecanismos no reestablecen el estado de forma precisa en la instrucción correspondiente tras una excepción o un fallo de predicción de saltos, viéndose obligados a reejecutar parte del código tras recuperar el estado en un punto previo a la instrucción. Este artículo se centra en la propuesta de un nuevo mecanismo de *checkpointing* para procesadores con consolidación fuera de orden que permita reestablecer el estado de forma precisa en cualquier instrucción y, al mismo tiempo, sea escalable para ventanas de instrucciones grandes. Dicho de otra manera, se proveerá al procesador con un mecanismo de *checkpointing* capaz de crear un *checkpoint* por instrucción sin que esto suponga un incremento sustancial en el hardware necesario y en el tiempo de acceso a las estructuras implicadas.**

Palabras clave—**consolidación fuera de orden (*out-of-order commit*), *checkpoint*, recuperación (*recovery*), renombrado (*renaming*).**

I. Introducción

La creciente diferencia de velocidad entre el procesador y la memoria es un problema importante que limita el rendimiento de los procesadores, ya que las instrucciones de acceso a memoria presentan latencias cada vez más grandes. Para aliviar este problema se intenta incrementar el paralelismo a nivel de instrucción, manteniendo el mayor número posible de instrucciones en vuelo para solapar los accesos a memoria con la ejecución de trabajo útil.

Los mecanismos presentes en procesadores actuales permiten mantener cientos de instrucciones en vuelo, las cuales sólo actualizan el estado de la máquina cuando se garantiza que la instrucción ha sido ejecutada correctamente y ninguna instrucción anterior puede invalidar su resultado. Para ello, el procesador consolida cada instrucción en orden del programa. No obstante, estos mecanismos no son escalables, es decir, permiten mantener en vuelo menos instrucciones de las que serían necesarias para ocultar totalmente la latencia de los accesos a memoria. Por este motivo, existen nuevas propuestas que utilizan mecanismos basados en *checkpoints* y permiten la consolidación fuera de orden. Un *checkpoint* es una estructura hardware que contiene toda la información necesaria para poder reestablecer el estado del procesador. Normalmente, esta información consiste en el mapeo de registros físicos y en la cola de registros físicos libres.

Reestablecer el estado de la máquina en la instrucción causante de una excepción o en un salto incorrectamente predicho está condicionado por el número de *checkpoints* que posee el procesador. Así, en el caso límite de que cualquier instrucción sea sensible de provocar una recuperación del estado del procesador, se necesitarían tantos *checkpoints* como instrucciones hayan en vuelo.

Sin embargo, disponer de un número muy grande de *checkpoints* no es sencillo, no sólo por el área necesaria y el consumo de energía que ocasiona, sino también porque el acceso a estas estructuras se efectúa dentro del camino crítico del procesador. Es por ello que para mantener un mayor número de instrucciones en vuelo, estos modelos relajan la precisión de la recuperación usando un número reducido de *checkpoints*. En caso de una excepción o un fallo de predicción, el procesador recupera el estado en el *checkpoint* inmediatamente anterior a la instrucción responsable, reejecutando por tanto algunas instrucciones, posteriores a la causante de la recuperación, que ya habían sido ejecutadas correctamente.

En este artículo proponemos un mecanismo para soportar recuperaciones precisas ante una excepción o fallo de predicción, lo que permitirá evitar el desperdicio de recursos y energía que supone esta reejecución de instrucciones. Además, como el coste hardware de la propuesta es reducido, el tiempo de ciclo no se vería afectado.

II. Descripción del problema

Los procesadores con consolidación fuera de orden, tales como el *CPR* [1] y los *kilo-instruction processors* [2, 3], mantienen un gran número de instrucciones en vuelo gracias a mecanismos que permiten liberar los recursos tan pronto como las instrucciones finalicen su ejecución.

Cada vez que se establece un nuevo *checkpoint*, la información del estado de la máquina es almacenada en una entrada de un vector de estructuras hardware con el fin de reestablecer dicho estado en caso de que haya que realizar una recuperación. Desde el momento en que se pueda garantizar que las instrucciones entre el *checkpoint* más antiguo y el inmediatamente posterior se han ejecutado correctamente y no serán descartadas a causa de excepciones o fallos de predicción, el *checkpoint* más antiguo podrá liberarse, permitiendo la consolidación de gran cantidad de instrucciones (cientos de ellas según el caso) de forma simultánea.

En estos mecanismos, la información del estado de la máquina es almacenada en las entradas de estas estructuras sin tener en cuenta ningún tipo de correlación entre estados, estableciéndose un control independiente y automático cada vez que la unidad de control tome la decisión de establecer un nuevo *checkpoint*. Se dará con cierta frecuencia el caso de que no será posible reestablecer el estado en la instrucción desencadenante de un proceso de recuperación debido a que no hay tantos *checkpoints* como instrucciones hay en vuelo. Esta falta de precisión en las recuperaciones constituye uno de los factores no deseables de estos mecanismos, pues cuanto mayor sea la distancia entre el *checkpoint* y la instrucción que se debe recuperar, más código debe ser reejecutado de forma innecesaria, degradando del rendimiento del procesador.

Para permitir recuperaciones precisas, el modelo que se presenta en este artículo se basa en la idea de que cuanto menor sea la distancia entre las instrucciones en la que se establecen *checkpoints*, mayor es la correlación entre los estados almacenados, es decir, comparten más información en común. Esto hace que pueda existir mucha información redundante en *checkpoints* consecutivos. Llevando este principio al caso extremo, si el procesador realiza un *checkpoint* por instrucción la variación puede ser como máximo igual al número de registros destino de la instrucción en cuestión (normalmente uno o ninguno).

Para poder establecer *checkpoints* para todas y cada una de las instrucciones en vuelo sin utilizar estructuras hardware excesivas que puedan perjudicar el camino crítico del procesador o incrementar en gran medida el coste de implementación, hemos seguido los siguientes criterios de diseño:

1. El almacenamiento del estado por cada instrucción en vuelo no puede establecerse de forma independiente debido al alto coste hardware que ello supondría. Para esto debe evitarse el almacenamiento de información redundante, reduciéndose el hardware necesario.
2. Es necesario establecer una estructura control que permita el renombrado libre de dependencias falsas y el control de tantos *checkpoints* como instrucciones haya en vuelo.
3. La estructura hardware de control debe permitir realizar recuperaciones del estado en la instrucción causante de una excepción o un fallo de predicción en un procesador con ejecución y graduación fuera de orden.

III. Modelo Algorítmico

Los procesadores con consolidación fuera de orden establecen criterios para establecer *checkpoints* a lo largo del código según se va ejecutando. Estos *checkpoints* se establecen en instrucciones con una alta probabilidad de que pueda ser necesario recuperar su estado, como por ejemplo un salto que resulta incorrectamente predicho con frecuencia o una instrucción susceptible de causar una excepción. Por ejemplo, CPR [1] usa técnicas para estimar si la predicción de un salto tiene un alto grado de fiabilidad. Para ello incluyen un estimador que determina la fiabilidad para cada uno de los saltos que se ejecutan, estableciendo un nuevo *checkpoint* en caso de baja fiabilidad.

Para realizar una recuperación en la instrucción causante, el procesador deberá eliminar todas las instrucciones posteriores a esa instrucción y reestablecer el mapeo de registros lógicos a físicos y la cola de registros físicos libres en ese punto. Una vez recuperado el estado, las siguientes instrucciones que se carguen harán uso de la información de este estado recuperado así como de los siguientes estados de la máquina que se vayan guardando.

La Tabla I muestra un ejemplo de recuperación. En esta tabla se muestran cinco instrucciones en secuencia numeradas por orden cronológico del programa:

TABLA I

CÓDIGO EJEMPLO.

```
            1.    store   r2    @
            2.    add     r1+r2   r2
            3.    bne     r2, #etiqueta
            4.    mov     r2    r1
            5.    add     r1+r2   r2
                  :
#etiqueta:
```

Cuando la instrucción 3 es decodificada, puesto que es un salto, se guardará el estado del procesador en un *checkpoint* por si se tuviese que recuperar más adelante. En el caso de que se detecte que la instrucción de salto 3 ha sido predicha incorrectamente, la instrucción 4 y siguientes hasta *#etiqueta* deben ser eliminadas del procesador, así como los posibles estados de la máquina almacenados por estas instrucciones. Las instrucciones que entren después de la recuperación usarán la información de mapeo de registros y cola de registros físicos libres recuperada desde el *checkpoint* que se hizo en la instrucción 3.

Nuestra propuesta mantiene en cada ciclo tantos estados como instrucciones hayan en vuelo, diferenciándose cada uno del anterior, como mucho, por la variación del mapeo de un registro lógico en uno físico. A pesar de la poca diferencia entre ambos estados, es necesario gestionar de forma explícita el control de asignación y liberación así como la recuperación de cada estado.

De esta manera, hemos establecido una serie de precondiciones para satisfacer los criterios de diseño:

1. Como hemos definido anteriormente, la diferencia entre estados consecutivos queda establecida por la variación de al menos uno de los registros, es decir, fijado un estado, la transición de estado queda establecida por la instrucción inmediatamente siguiente al estado primeramente establecido, donde el registro destino que pudiera tener dicha instrucción constituiría la información diferencial.
2. Como modelo en primera aproximación, se debe establecer una información adicional de control. El secuenciamiento de las instrucciones del programa que se esté ejecutando será de forma explícita, pues desde el momento en que las instrucciones entran a la ventana de instrucciones, su emisión, ejecución y graduación se realiza fuera de orden, sin el perjuicio de que en caso de recuperación no se pueda identificar la instrucción concreta en la cual se tiene que reestablecer el estado.
3. La información común entre estados no puede duplicarse, por lo que debe establecerse un control que identifique para cada elemento de información – valor de registro – cual es el conjunto de estados al que pertenece.

Para cumplir el primer punto, se modificará la filosofía del *checkpoint*. En vez de guardar el estado general de todos los mapeos entre registros lógicos y físicos, solamente se guardará información para el registro lógico destino de la instrucción actual. De esta forma, se evita almacenar información redundante entre *checkpoints* consecutivos. Los *checkpoints* se construyen de forma incremental, teniendo en cuenta todos los anteriores.

Para cumplir el segundo punto se establece una etiqueta contadora de secuencia por cada instrucción. Dada una instrucción que produzca una recuperación del estado de la máquina debido, por ejemplo, a un fallo en la predicción, el control de recuperación atenderá al valor de la etiqueta contadora asociado a la instrucción que ha producido el fallo. El control mantendrá en vuelo todas aquellas instrucciones que tengan un valor de etiqueta contadora menor a la de la instrucción que ha producido fallo en la predicción y descartará todas aquellas que tengan una etiqueta mayor o igual a dicha instrucción. Si el procesador dispusiera de varias unidades para resolver los saltos, en caso de producirse una recuperación por más de un salto resuelto en dicho ciclo, se tomará el valor de etiqueta contadora para el control de recuperación de aquel salto más antiguo, es decir, el de menor valor entre los saltos resueltos y mal predichos.

Para cumplir el tercer punto se debe establecer una lista de los identificadores de estado al cual pertenece un elemento de información almacenado, entendiendo por elemento de información el valor del registro físico asociado al registro lógico en dicho punto del programa. Dado que cada instrucción establece un nuevo estado, podemos simplificar el diseño haciendo coincidir los identificadores de estado con la etiqueta contadora de cada instrucción.

De esta manera, cada elemento de información estará asociado con dos valores identificadores de estado, estableciendo un *rango de estados*: el valor inferior se corresponderá con el valor de etiqueta contadora de la instrucción actual y el valor superior será el valor de la etiqueta contadora de la instrucción anterior a la próxima instrucción que redefina el registro lógico. Dado que a cada registro físico – elemento de información – se le asocia un rango de estados, el proceso de recuperación de estado sólo tendrá que reestablecer la información correspondiente a aquellos registros físicos cuyo rango de estados contenga el valor de la etiqueta contadora de la instrucción que provoca la recuperación.

Atendiendo al ejemplo de la Tabla I, si partimos de un estado inicial identificado con la etiqueta *0*, los diferentes identificadores de estado asociados a cada instrucción quedan según la Tabla II.

TABLA II

IDENTIFICADOR DE ESTADO ASOCIADO A CADA INSTRUCCIÓN PARA EL CÓDIGO EJEMPLO DE LA TABLA I.

		Id estado		
	1.	*0*	store	r2 @
	2.	*1*	add	r1+r2 r2
	3.	*1*	bne	r2, *#etiqueta*
	4.	*2*	mov	r2 r1
	5.	*3*	add	r1+r2 r2
		:		
#etiqueta:				

Las instrucciones 1 y 3 no constituyen una variación del estado de los registros: los stores constituyen variaciones del estado de la memoria y no se tienen en cuenta porque se gestionarán de forma independiente y los saltos sólo implican un cambio en el flujo de ejecución. Las instrucciones 2 y 5 constituyen un cambio de estado debido a una variación del valor del registro lógico r2, mientras que la 4 realiza un cambio sobre el registro lógico r1. La instrucción 3 constituye la instrucción de salto que producirá una recuperación.

El rango de estados asociados a cada registro físico queda de la forma que se muestra en la Tabla III.

TABLA III

RANGOS DE ESTADOS ESTABLECIDOS PARA CADA REGISTRO FÍSICO SEGÚN EL CÓDIGO EJEMPLO DE LA TABLA I Y II.

Rangos de Estados		Registro asociados	
Inferior	Superior	Lógico	Físico
0	0	R2	R2.0
1	2		R2.1 *
3	3		R2.2
0	1	R1	R1.0 *
2	3		R1.1

Para la Tabla III se ha considerado la notación R*x.y* como el registro físico *y* asociado al registro lógico *x*. Se observa que por cada reasignación sobre el mismo registro lógico se establece un nuevo registro físico,

eliminando así cualquier falsa dependencia. Los valores R1.0 y R2.0 constituyen los valores previos asociados a los registros lógicos r1 y r2 respectivamente antes de la ejecución del código ejemplo.

Tras ejecutarse la instrucción 3 y detectarse el fallo en la predicción, se debe establecer como último estado válido el fijado por la instrucción 3, es decir, el estado *1*. Por tanto, los valores del estado recuperado (el mapeo de registros físicos en registro lógicos) serán los correspondientes al intervalo en los que esté incluido el estado asociado a la instrucción 3, estado *1*, indicados con el símbolo * en la Tabla II. La etiqueta contadora de la instrucción 3 que desencadena la recuperación precisa determina qué valor físico para cada registro lógico debe reestablecerse para poder continuar la ejecución. Todos aquellos registros físicos cuyo valor de rango inferior sea superior al estado de la instrucción 3 podrán ser liberados, pasando a ser registros físicos reutilizables en el renombrado para futuras instrucciones que se decodifiquen.

Una vez realizado el proceso de recuperación, el contador global que define los valores de las etiquetas contadoras debe reestablecerse al valor del estado recuperado con el fin de proseguir el secuenciamiento de los nuevos estados de acuerdo con el valor del último estado válido. Cabe decir que en caso de recuperación por excepción, el estado a recuperar será el asociado a la instrucción que causa la excepción menos uno.

IV. Modelo de Implementación

Definido el algoritmo para recuperar el estado en una instrucción concreta dentro de un procesador con consolidación fuera de orden, el principal objetivo de la implementación es no penalizar el tiempo de ciclo de la etapa de renombrado. En esta etapa se realiza el mapeo de registros lógicos a físicos, controlándose también la asignación y liberación de los registros físicos ya sea por reasignación o por recuperación, así como el establecimiento de los estados de la máquina.

La Figura 1 muestra las estructuras de control necesarias, por registro lógico, para la implementación del mecanismo. Además, para simplificar aún más el mecanismo, se divide el banco de registros físicos, asignando permanentemente un conjunto de registros físicos a un lógico. Las estructuras de control son las siguientes:

- Vector de asignación (*hot-bit pointer*), cada campo del cuál es un bit. Este vector implementa un puntero de posicionamiento cíclico (un bit es 1 y el resto es 0) el cual determina el último mapeo de ese registro lógico a un físico. Constituye en sí el control de renombrado de registros.
- Dos vectores constituidos por las etiquetas contadoras. Uno de estos vectores (*lower tag*) constituye el límite inferior del rango de estados asociado al mapeo de ese registro lógico a físico, y el otro vector (*upper tag*) constituye el límite superior. El límite inferior se inicializa con el valor de la etiqueta contadora de la primera instrucción que modificó ese registro lógico como registro destino. El límite superior se va modificando con el valor de las etiquetas contadoras de las siguientes instrucciones que no cambian el mapeo de ese registro lógico, es decir, que tienen otro registro lógico como destino.
- Un comparador de rangos de estado de la máquina. Esta estructura actuará sobre el vector de asignación para establecer un nuevo índice en caso de recuperación. Este es un proceso que posiciona el *hot-bit pointer* al valor físico del registro lógico en cuestión, siendo definido por el estado al que se desea recuperar (*recovery tag*). En el proceso de recuperación se activará solamente aquel vector de posicionamiento del registro físico de forma que *recovery tag* esté dentro del rango [*lower tag* .. *upper tag*].

La referencia al registro físico mapeado se convierte, en vez de ser un identificador de registro, en la pareja <*registro lógico, posición del hot-bit pointer*>.

A. La liberación de registros físicos

Una de las causas de la degradación del rendimiento de un procesador puede ser una mala elección del tamaño del banco de registros físicos. En el caso de que todos los registros físicos estén asignados, la etapa de búsqueda y decodificación de instrucciones se tiene que parar puesto que no se pueden hacer más mapeos entre registros lógicos y físicos. Esta parada finaliza cuando se libera algún registro físico.

Atendiendo al diseño propuesto, este problema se acrecienta puesto que el banco de registros físicos del procesador está dividido en varias partes, de forma que se asigna permanentemente un conjunto de registros físicos al mismo registro lógico. Esto puede hacer que el *front-end* del procesador se pare con más frecuencia puesto que ahora el número de mapeos "vivos" del mismo registro lógico está más limitado.

Para aliviar este problema, se ha adoptado una técnica sencilla para anticipar la liberación de los registros físicos, que es fácilmente aplicable a ventanas de instrucciones grandes en procesadores con graduación fuera de orden. Esta técnica consiste en liberar el registro físico tras ser leído por la última instrucción dependiente. Para ello es necesario establecer un bit de control (*ready bit*) para cada registro físico y modificar la matriz de dependencias. El *ready bit* es un bit que indica la disponibilidad del un valor del registro físico al cual esta asociado. Determinará cuando el valor del registro estará disponible para ser leído.

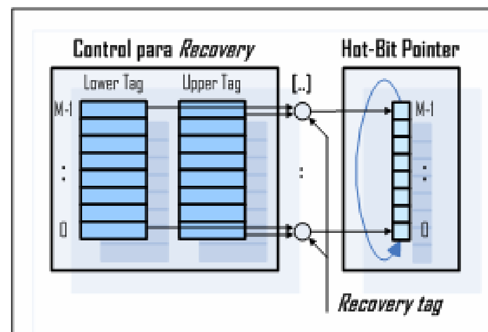

Fig. 1. Estructuras de control asociadas a cada registro lógico.

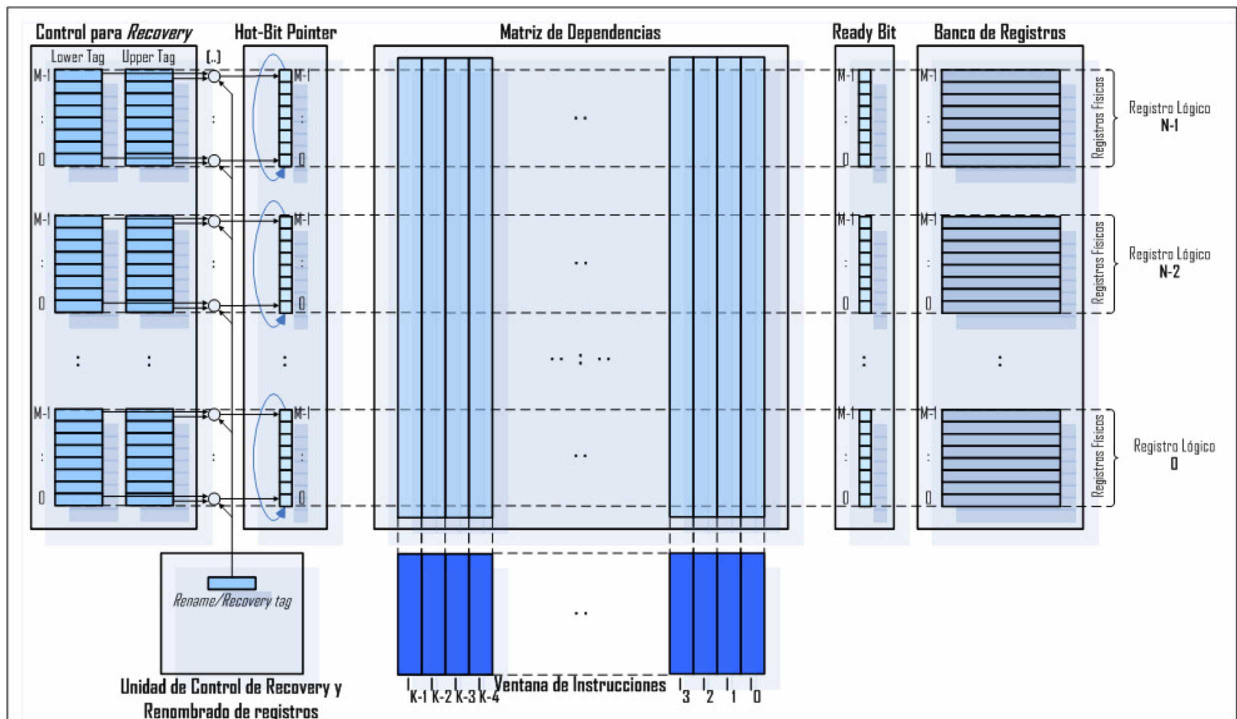

Fig 2. Diseño de implementación como primera aproximación. No se ha especificado el diseño de las etapas de ejecución y colas load/store.

La matriz de dependencias [11], en principio, está diseñada para saber cuando una instrucción tiene todos sus operandos fuente calculados y puede, por tanto, ser ejecutada. Además, con esta matriz, se puede saber qué instrucciones, dentro de la ventana de instrucciones, están aún pendientes de leer un registro físico determinado. De esta forma, un registro físico se puede liberar en el momento que no haya ninguna instrucción pendiente de leer el valor de este registro y, además, la etiqueta contadora del último estado que se sabe que es válido, sea superior al valor del límite superior del rango de estados de este registro.

En la Figura 2 se muestra el diseño general de estas dos estructuras.

La matriz de dependencias interactúa con la ventana de instrucciones y con el vector de asignación de cada registro lógico para determinar si algún registro físico de un lógico dado, puede ser asignado como registro destino. En el caso de que haya algún registro físico libre, esto se sabe mediante la matriz de dependencias, el vector de asignación incrementará su posición apuntando al nuevo registro físico mapeado. En el caso de que no se disponga de ningún registro físico libre, se procederá a detener las etapas de búsqueda y decodificación de instrucciones.

V. Trabajos Relacionados

Hwu y Patt [4] propusieron el uso de *checkpoints* para recuperar el estado de la máquina en las instrucciones causantes de excepción o en las predichas de forma incorrecta. Smith y Pleszkun [7] propusieron implementaciones alternativas, como los *buffers* de historia. Sin embargo, el propósito de estos trabajos no es eliminar la necesidad de graduar las instrucciones en orden sino utilizar *checkpoints* para restaurar el estado de la máquina. El Pentium 4 [10] utiliza una tabla de alias para hacer el mapeo de los registros liberados mientras que el MIPS R10000 [9] y Alpha 21264 [5] usan el mecanismo de *checkpoints* para recuperar el mapeo de los registros renombrados. El método basado en *checkpoints* utilizado por el MIPS R10000 y Alpha 21264 no es aplicable a ventanas de instrucciones grandes por falta de escalabilidad. El Power4 de IBM [8] mantiene una ventana de instrucciones más grande agrupando hasta seis instrucciones por cada entrada, aunque este crecimiento es sólo lineal.

Cherry [6] utiliza un único *checkpoint* para liberar recursos con antelación cuando puede garantizarse que todos los saltos han sido completados y todas las operaciones de memoria han sido emitidas. La liberación anticipada de recursos permite aumentar el número de instrucciones en vuelo hasta cierto punto.

Los *kilo-instruction processors* [2, 3] permiten aumentar todavía más el número de instrucciones en vuelo. Esta propuesta establece *checkpoints* en loads de larga latencia y a determinados intervalos periódicos, permitiendo emular una ventana de instrucciones virtual con un gran número de entradas y liberar los recursos rápidamente.

Otra propuesta es el CPR [1] que, de forma similar, usa *checkpoints* para aumentar virtualmente el tamaño de la ventana de instrucciones. Tanto el CPR como los *kilo-instruction processors* presentan una solución completa y escalable basada en *checkpoints* selectivos para diseñar un procesador capaz de mantener una gran cantidad de instrucciones en vuelo.

VI. Resumen

Nuestra propuesta abarca el principal objetivo de los procesadores con consolidación fuera de orden: la

escalabilidad del diseño para soportar un gran número de instrucciones en vuelo. Sin embargo, mejoramos los diseños previos al permitir recuperaciones precisas del estado de la máquina en las instrucciones que causan excepciones o han sido predichas de forma incorrecta.

El diseño preliminar presentado en este artículo constituye nuestra primera aproximación en esta línea de investigación. Aunque algunos resultados preliminares nos hacen pensar que hay buenas expectativas, estamos actualmente realizando un extenso análisis y evaluación de nuestra propuesta en comparación con mecanismos representativos del estado del arte con el objetivo de presentar un estudio más completo.

VII. Agradecimientos

Este trabajo se ha realizado gracias a la financiación de la Universitat Politècnica de Catalunya, por medio de la beca predoctoral UPC para investigación, y gracias al apoyo del Ministerio español de Ciencia y Tecnología mediante el contrato TIN-2004-07739-C02-01, así como de la red europea de excelencia HiPEAC y el Barcelona Supercomputing Center - Centro Nacional de Supercomputación. También deseamos dar las gracias a Adrián Cristal, José María Llabería, Marco Galluzzi y Tanausú Ramírez por sus explicaciones y comentarios.

VIII. Referencias

[1] H. Akkary, R. Rajwar, and S. T. Srinivasan. Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors. *36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-36), 2003.*

[2] A. Cristal, D. Ortega, J. Llosa, and M. Valero. Out-of-Order Commit Processors. *10th International Symposium on High Performance Computer Architecture (HPCA'04), 2004.*

[3] A. Cristal, O. J. Santana, M. Valero, and J. F. Martínez. Toward kilo-instruction processors. *TACO, 2004.*

[4] W. W. Hwu and Y. N. Patt. Checkpoint repair for out-of-order execution machines. *14th Annual International Symposium on Computer Architecture, June 1987.*

[5] D. Leibholz and R. Razdan. The Alpha 21264: A 500 MHz out-of-order execution microprocessor. *42nd IEEE Computer Society International Conference (COMPCON), February 1997.*

[6] J. F. Martínez, J. Renau, M. C. Huang, M. Prvulovic, and J. Torrellas. Cherry: Checkpointed early resource recycling in out-of-order microprocessors. *35th International Symposium on Microarchitecture, November 2002.*

[7] J. E. Smith and A. R. Pleszkun. Implementation of precise interrupts in pipelined processors. *12th Annual International Symposium on Computer Architecture, June 1985.*

[8] J. M. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy. POWER4 system microarchitecture. *IBM Journal of Research and Development, January 2002.*

[9] K. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro, April 1996.*

[10] G.Hinton, D.Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal, February, 2001.*

[11] M. Goshima, K. Nishino, Y. Nakashima, S. Mori, T. Kitamura, and S. Tomita. A high-speed dynamic instruction scheduling scheme for superscalar processors. *34th annual ACM/IEEE international symposium on Microarchitecture, 2001.*