Simulación rápdia de procesadores

Memòria del Projecte Fi de Carrera
d'Enginyeria en Informàtica
realitzat per
Oscar Vila Ramírez
i dirigit per
Juan Carlos Moure López
Bellaterra 19.de..Setembre.de 2007.

Universitat Autònoma de Barcelona

Escola Tècnica Superior d'Enginyeria

Informe del Director

Enginyeria en Informàtica

Alumne: Oscar Vila Ramírez

Títol del projecte: Simulación rápida de procesadores

Director del projecte: Juan Carlos Moure Lopez

Departament/Unitat: Unitat d'Arquitectura d'Ordinadors

Comentaris sobre l'acompliment dels objectius i del rendiment:

Observacions:

Valoració (nota) del director:

Signatura del director

Bellaterra, ... de .....de .....

Índice

Me gustaría hacer una mención especial para dedicar la realización de este proyecto a una persona muy especial para mí. Jose Ángel Pérez Gómez, fallecido en accidente de tráfico el día 27 de Julio de 2007.

Descansa en paz amigo.

1. Objetivos y motivaciones

Cuando se diseña un nuevo procesador se tiene que comprobar su rendimiento. Para mejorar el rendimiento del procesador se puede modificar el número de sus componentes, el tamaño de estos componentes, como el tamaño de tablas o memorias, su velocidad, y las conexiones entre los componentes. Realizar estas modificaciones saldría muy caro si se hiciera mediante hardware, de ahí que se utilicen simuladores. Con el uso de simuladores se puede reducir mucho el costo del diseño del procesador. Por ejemplo, la simulación permite analizar el comportamiento de la memoria caché, predictor de saltos, entre otros componentes.

El coste del procesador viene dado por varios parámetros: velocidad, área que ocupa cada componente, valor del material empleado y consumo energético. Cabe recordar que los procesadores tienen un tamaño limitado y dentro tienen que caber todos los componentes. Por ejemplo, un predictor de saltos muy pequeño es barato y consume poco, y el tiempo de acceso es muy bajo. El problema que tiene un predictor de saltos pequeño es que guarda muy poca historia de los saltos anteriores y por ello la precisión de la predicción es muy baja. Un predictor de saltos grande mejora la capacidad de guardar un histórico de sus predicciones con lo que puede hacer una mejor predicción para el salto que esta analizando, por el contrario, ocupa más área y aumenta su valor y consumo energético. En la figura 1.1.A se puede apreciar que conforme se aumenta el tamaño del predictor de saltos la tasa de acierto aumenta. El grafico de la figura 1.1.B muestra que el tiempo que tarda el predictor en dar la su predicción se incrementa según se aumenta su tamaño.

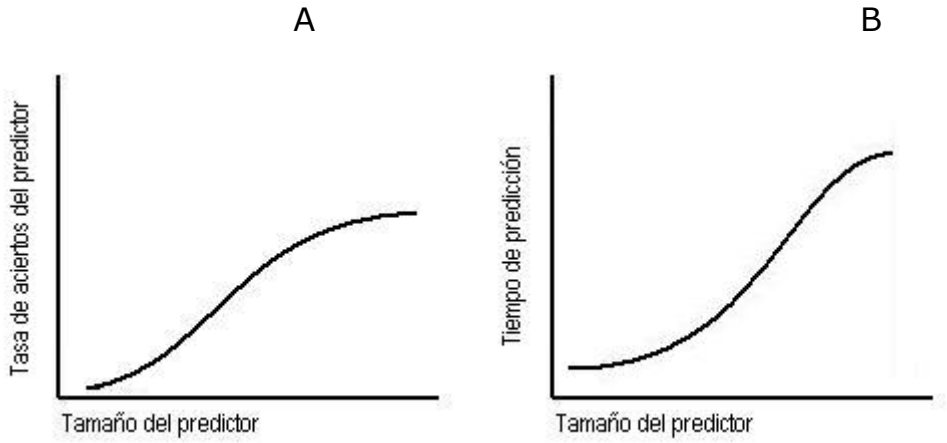


Figura 1.1: A – Gráfica que muestra la Tasa de aciertos en función del tamaño de un predictor
B – Grafica que muestra como varia el tiempo de predicción conforme cambia el tamaño del predictor.

El mundo de la informática es muy competitivo y avanza muy deprisa. Se necesitan resultados rápidos para poder sacar un producto antes que la competencia. Entonces nos aparece un nuevo problema, la duración de la simulación. Dado que tenemos que probar el procesador con varias configuraciones necesitamos que la ejecución de la simulación produzca sus resultados lo más rápidamente posible.

Tal y como se muestra en la figura 1.2, hay dos sistemas que considerar, el Sistema Real y el Sistema Simulado. En el sistema real se ejecuta el simulador y en el sistema simulado se ejecutan los programas de prueba. Cada uno de los sistemas determina un dato que necesitamos para realizar el estudio. El sistema real nos da los tiempos de ejecución de las simulaciones. Del sistema simulado obtendremos resultados de la ejecución del procesador, como el número de fallos del predictor de saltos, o numero de fallos en la memoria cache. Estos datos serán utilizados para guiar el diseño del procesador.

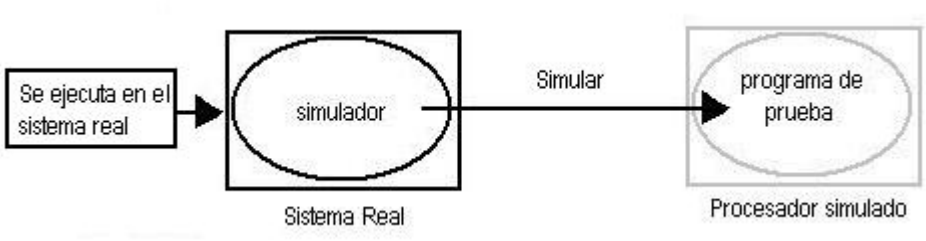


Figura 1.2: Esquema que muestra dos sistemas, uno real y un simulado.

En este proyecto se usará el del predictor de saltos de un procesador como ejemplo de proceso de diseño. Este estudio requiere de simular diferentes predictores con diferentes parámetros. El objetivo es realizar las diferentes simulaciones lo más rápido posible obteniendo unos resultados de las simulaciones lo más cercanos posible a la realidad. Para poder reducir tiempo en la simulación primero se debe dividir la ejecución en partes que llamaremos fragmentos. De todos los fragmentos de la simulación se deberá hacer una selección para determinar cuales son significativos para el resultado final. Los fragmentos de la simulación que se hayan escogido como determinantes serán los que se simularan con todas las configuraciones. De este modo se podrá reducir el tiempo de simulación ya que no se simula todo el programa, únicamente los fragmentos relevantes.

Excluir fragmentos en la simulación de una aplicación provoca que el resultado que se obtiene de la simulación sea diferente del resultado real. En la figura 1.3 se puede observar como el resultado varía según aumenta el tiempo de simulación, es decir, cuantos mas fragmentos se simulan menos errores se comete.

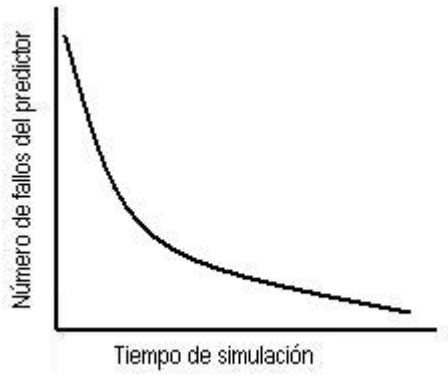


Figura 1.3: La gráfica muestra el número de fallos que comete un predictor en función del tiempo de simulación.

Existen varias estrategias para poder reducir el tiempo en la simulación de los programas, nosotros hemos escogido una de ellas que describiremos a continuación.

La simulación se divide en fragmentos. Primero se simula un conjunto de fragmentos escogidos de forma homogénea (equidistantes). Después se simulan más fragmentos escogidos también de forma homogénea. De esta manera que tratamos de que los fragmentos escogidos sean representativos de toda la simulación.

La figura 1.4 muestra un ejemplo de cómo se escogerían los fragmentos de la simulación para intentar reducir el tiempo de simulación. Si únicamente simulamos las muestras y omitimos le resto del programa el tiempo quedaría reducido considerablemente. El resultado final seria próximo al valor de simular el programa completamente (línea punteada).

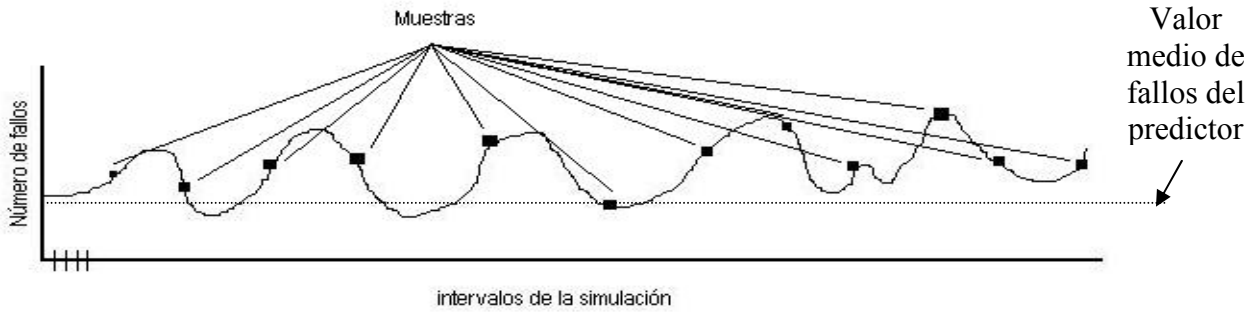


Figura 1.4: Gráfica que muestra una simulación de una aplicación y como se escogen las muestras a ser simuladas para reducir el tiempo de simulación.

Describamos el algoritmo para entenderlo:

1. Determinar un error máximo.
2. simular un fragmento en la primera pasada (R1)
3. simular el doble de fragmentos que la iteración anterior (R2)
4. calcular el error entre las 2 ultimas iteraciones (Diferencia R1, R2)
   4.1. si la diferencia es mayor al error máximo R2=R1 y vamos a 2
   4.2. si es menor o igual al error máximo hemos terminado

La ventaja principal de este mecanismo es que no es necesario simular todo el programa completo, y que se dispone de un mecanismo para estimar el error que cometemos al no realizar la simulación completa. En todo momento podemos decidir si la estimación de error es suficiente y parar la simulación, o si deseamos continuar para corroborar los resultados. La desventaja es que quizá los fragmentos que se hayan escogido no sean los más significativos y el resultado diste más de lo que se espera de la realidad.

Para la realización de este proyecto se simulará la ejecución de un procesador similar al Alpha 21264. Esta simulación se realizará bajo entorno Linux utilizando la herramienta SimpleScalar.

Para mejorar los tiempos de simulación de las pruebas también se propondrá una extensión a nuestra propuesta. Nuestro proyecto esta realizado en un sistema monoprocesador, de modo que una maquina simula todas las pruebas. La extensión consiste en utilizar varios procesadores de forma simultánea para realizar las simulaciones, reduciendo así aun más el tiempo de simulación para obtener el resultado final.

En los posteriores temas se expondrán una serie de conceptos previos necesarios para la comprensión del estudio. Se expondrán una serie de palabras técnicas que se emplearan mas adelanta y se explicara el simulador con todos sus componentes y las configuraciones posibles de cada uno. Se explicará también las aplicaciones que componen cada benchmark.

Seguidamente se expondrán una serie de propuestas para la resolución del problema. De todas las propuestas que se mostraran

se escogerá una que posteriormente será analizada en profundidad.

En el siguiente tema se realizara un pequeño estudio previo para terminar de perfilar los parámetros del estudio. Servirá también para familiarizarnos con el simulador y ver su funcionamiento. Además se mostrarán una serie de problemas acaecidos durante la familiarización con el entorno y como se han resuelto.

Para finalizar se mostrarán los resultados del estudio de forma gráfica. Se explicarán los resultados y se analizarán. Después de mostrar los resultados con las aplicaciones de prueba se realizará otro estudio con otras aplicaciones de un SPEC superior. Estos resultados también serán mostrados y analizados debidamente. Para terminar el estudio se propondrá una mejora de simulación en paralelo.

2. Conceptos previos

En este tema se describirá el funcionamiento de un procesador. Explicaremos que es y como funciona un procesador segmentado y las ventajas y desventajas que tiene. El predictor de saltos es el componente del procesador en el que se centrará esta explicación. Del predictor de saltos explicaremos que es y como funciona.

2.1. El Procesador

El procesador es el encargado de ejecutar las instrucciones de una aplicación. Para ejecutar cada una de las instrucciones el procesador se guía por una serie de pasos a seguir. Primero busca la instrucción en memoria. Luego descodifica la instrucción para saber que es lo que tiene que hacer. En un tercer paso se ejecuta la instrucción. Después se guarda el resultado de la ejecución en memoria y finalmente se finaliza la ejecución. Estos pasos los describiremos como IF (instruction fetch) búsqueda de instrucción, ID (instruction decode) descodificación de instrucción, EX (Execute) ejecución, MEM (memory) acceso a memoria y WB (write back) para escribir el resultado. Un procesador que contiene este conjunto de etapas recibe el nombre de procesador segmentado.

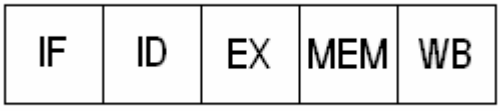


Figura 2.1: Esquema que muestra el orden que mantiene un procesador segmentado.

Las instrucciones de cualquier aplicación van numeradas y su ejecución, en la mayoría de casos es secuencial. Para llevar la cuenta de la instrucción que se debe procesar se necesita un modulo que sepa que instrucción esta en curso. El modulo que indica la instrucción que esta en curso es el PC (program counter) contador de

programa. El PC lo único que hace es ir incrementándose de uno en uno. El número que indica el PC es el número de instrucción que se debe ir a buscar en la etapa IF del pipeline.

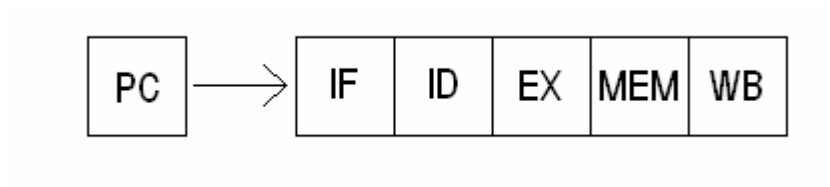


Figura 2.2: El contador de programas le indica a la primera etapa del procesador (IF) el número de instrucción siguiente que debe ir a buscar.

2.2. El procesador segmentado

Cada vez que un procesador procesa una instrucción debe seguir todos los pasos descritos en el apartado anterior y hasta que no finalice la instrucción en curso no se puede empezar con la siguiente. Aquí aparece un problema de desaprovechamiento de recursos. Mientras se tiene una instrucción en la etapa de ejecución, por ejemplo, el resto de etapas del procesador están sin utilizar esperando que llegue alguna instrucción para realizar su trabajo. Tal y como muestra la figura 2.3.

Figura 2.3: Muestra una instrucción cualquiera en la etapa 3 de un procesador y el resto de etapas están vacías esperando que se termine la ejecución para recibir la siguiente instrucción.

Para solventar el problema del desaprovechamiento de recursos se pueden ejecutar varias instrucciones al mismo tiempo. La forma de hacerlo es, mientras una instrucción está en la etapa ID, se va a buscar la siguiente instrucción que ha de ejecutarse, y así sucesivamente. La figura 2.4 muestra el funcionamiento que debería seguir el procesador.

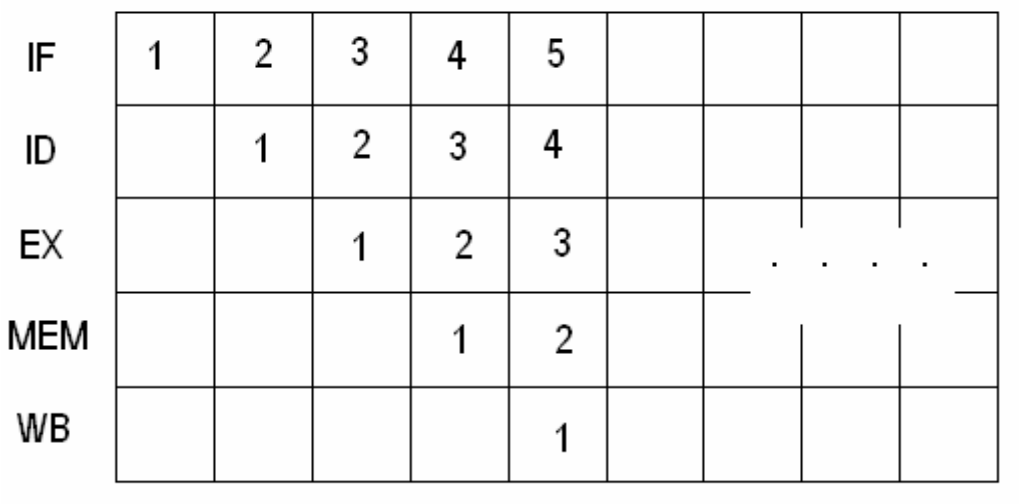


Figura 2.4: Muestra la entrada y ejecución de 5 instrucciones en un procesador segmentado de 5 etapas durante cinco ciclos.

En la figura 2.4 se muestra claramente la entrada de cinco instrucciones de forma consecutiva y su paso por las diferentes etapas del pipeline. Se puede comprobar que cuando entra la instrucción 5 en la etapa IF, la instrucción 1 va por la etapa WB, la 2 se encuentra en la etapa MEM, la 3 en la etapa EX y la 4 en la etapa de descodificación. Este tipo de procesador se le llama procesador segmentado. El procesador segmentado aprovecha mejor los recursos libres y hace que la ejecución sea mucho más rápida.

Con el procesador segmentado aparecen una serie de problemas. El primero de ellos serían las dependencias de datos entre instrucciones. Una dependencia sería, por ejemplo, una instrucción I obtiene como resultado de su ejecución un dato D, la instrucción N procesa el dato D para obtener su resultado. El problema vendría cuando las instrucciones I y N están lo suficientemente cerca como para que I no pueda obtener el resultado antes que N lo pretenda procesar.

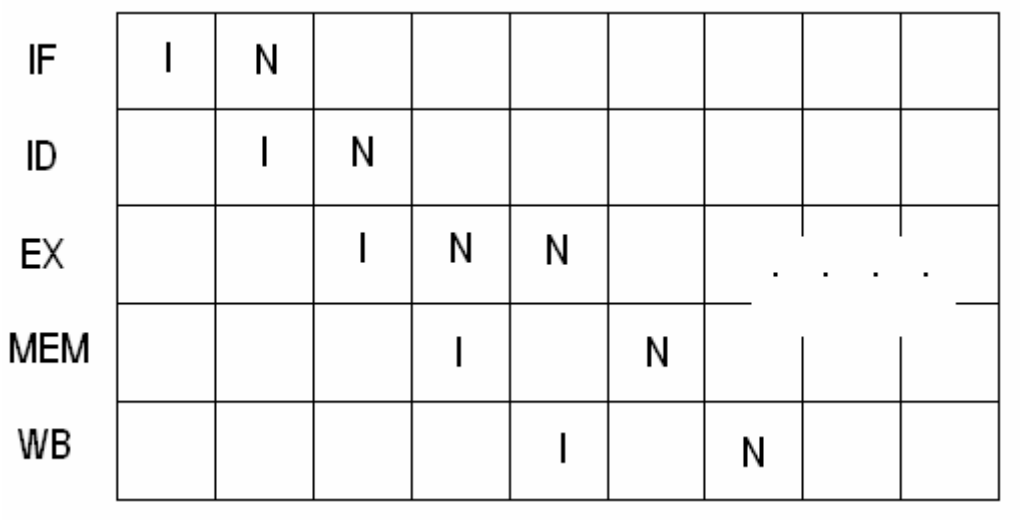


Figura 2.5: Ejecución de 2 instrucciones I y N. El resultado de N depende del resultado obtenido en I.

En la figura 2.5 se puede observar cómo la instrucción I se procesa de forma normal, mientras que la instrucción N queda estancada en la etapa ID y no puede acceder a memoria hasta que la instrucción I finaliza su ejecución. Con este problema perdemos un ciclo de procesamiento (en el caso que cada etapa se ejecute en un ciclo) y hace que se retrase el procesamiento de las instrucciones siguientes.

Otro problema que surge con el procesador segmentado son las instrucciones de salto. La instrucción de salto es un tipo de instrucción que permite variar el PC para desplazar la ejecución a otro punto del programa. Hay dos tipos de instrucciones de salto, el salto condicional y el salto incondicional. El salto incondicional siempre salta a la dirección que indica la instrucción, mientras que el salto condicional tiene que comprobar el resultado de una condición para decidir si salta o no. La ejecución de la instrucción de salto se realiza en la etapa 3. En el caso que la instrucción de salto fuera un salto incondicional o un salto condicional cuya condición diera como resultado que salta, el PC seria modificado con la siguiente instrucción a ejecutar y las instrucciones que hay en IF e ID no servirían.

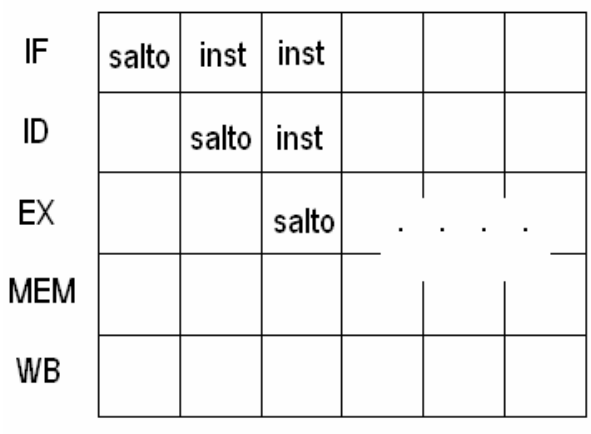


Figura 2.6: Muestra el comportamiento que sigue un procesador segmentado al entrar una instrucción de salto seguida de dos instrucciones cualquiera.

Como muestra en la figura 2.6, la instrucción de salto entra en el pipeline, y se ejecuta en el paso 3, y las 2 siguientes instrucciones que entran después deberían ser eliminadas del pipeline para empezar a procesar la primera instrucción que indica la instrucción de salto. Con este problema perdemos dos ciclos de reloj en cada salto. Para solventar este problema introduciremos un modulo llamado predictor de saltos. El predictor de saltos se encarga de predecir qué instrucción es la que debe entrar en el pipeline después que entre una instrucción de salto.

2.3. La BTB

Para poder utilizar el predictor de saltos se necesita una herramienta que permita saber de forma eficaz si la siguiente instrucción es un salto o no, esta herramienta se lama BTB (Branch Target Buffer).

Para introducir un predictor de saltos en un procesador se dispone de dos opciones. Si la instrucción se ejecuta en la etapa 3, el predictor deberá ir antes, en la 2 o en la 1. En la etapa 2 ya sabemos que es una instrucción de salto y donde debe saltar, lo que no sabemos es si debe saltar, por lo que es un buen lugar donde colocar

nuestro predictor de saltos. El problema de colocar el predictor de saltos en la etapa 2 es que perdemos un ciclo si el salto se predice, ya que en la etapa 1 del pipeline se habrá introducido la instrucción siguiente al salto. Lo ideal seria colocar el predictor de saltos en la etapa 1 del pipeline, pero esta etapa es una etapa de búsqueda, no sabemos nada sobre la instrucción. La solución pasa por registrar las instrucciones de salto en una tabla a parte, la Branco Target Buffer. La BTB también almacena el número de la instrucción donde se pretende saltar. Cuando una instrucción entra en el pipeline se va paralelamente al buffer a comprobar si es de salto o no, y al mismo tiempo a comprobar en el predictor si decidimos que salta o no. Veámoslo mejor en la figura siguiente.

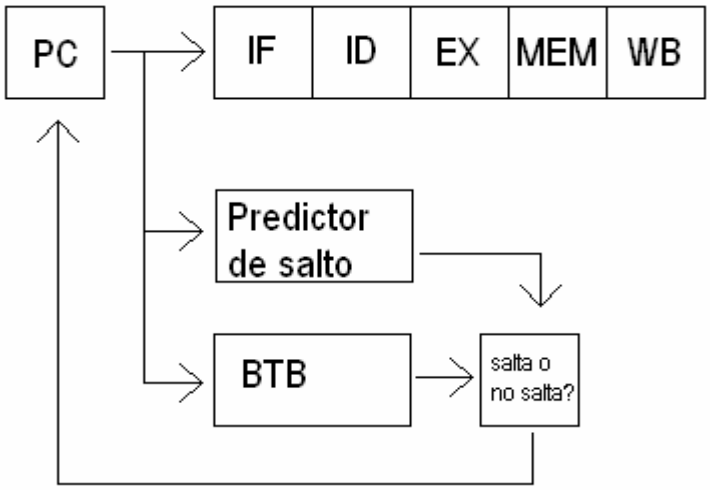


Figura 2.7: Lugar donde se debe colocar el predictor de saltos y la Branch Target Cache en un procesador segmentado.

En la figura 2.7 se puede comprobar que el número de instrucción que PC envía a IF también va a parar al predictor de saltos y a su vez a la BTB. En el caso que la instrucción que entra fuera un salto y nuestro predictor decidiera que salta, se modificaría el PC para que la siguiente instrucción que entre en el pipeline fuera la que corresponde al resultado del salto. De esta forma, si se acierta el salto no perdemos ciclos en la ejecución. Si el predictor de saltos falla en su decisión de salto, perderíamos dos ciclos.

2.4. Mecanismos para predecir saltos

Hasta el momento se ha podido ver dónde va y cómo es un predictor de saltos. La pregunta ahora es, ¿cómo funciona un predictor de saltos?

Vemos el caso más sencillo. En este caso el predictor de saltos predice siempre que el salto no se realiza, de esta forma las instrucciones que entran en el pipeline son las que siguen a la instrucción de salto de forma consecutiva. El problema es que si estamos en un bucle que se repite muchas veces, perdemos dos ciclos por cada vez que se repite el bucle. Para solventar este problema podemos crear un predictor de saltos de un bit. Este bit almacena el resultado de cada instrucción de salto. De esta forma, si se entra en un bucle solo fallará la primera y la ultima predicción. El predictor de un bit también tiene inconvenientes. Imaginemos que vamos entrando en bucles de forma sucesiva. Por cada uno de ellos fallamos dos veces, la primera vez que entramos y la ultima vez que es la de salida del bucle. Si colocamos un predictor de dos bits esto no sucedería. Supongamos que el 0 es que no salto la última vez y el 1 es que salto la última vez.

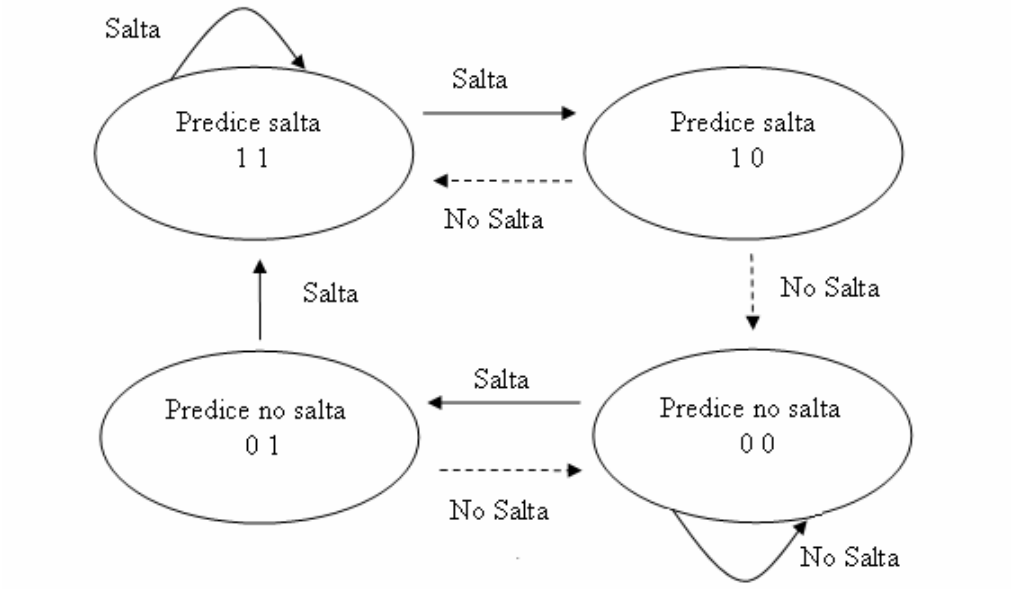


Figura 2.8: Esquema de funcionamiento de un predictor de saltos de 2 bits.

Según la figura 2.8, en cuanto se entre a un bucle de más de una iteración, nuestro predictor predecirá que salta. Cuando se salga del bucle, el predictor seguirá prediciendo que salta una vez más. De este modo cada dos fallos es cuando cambia su predicción. Este predictor de dos bits se puede convertir en un contador. El problema vendría cuando el contador es muy grande. En el caso en que entremos en un bucle muy grande, al salir del bucle el predictor seguirá prediciendo que salta en el resto de predicciones de la aplicación. Esto se puede cortar haciendo un contador negativos, o de fallos. El contador negativo contaría dos o tres fallos en la predicción y pondría el contador a 0 para volver a empezar.

Los predictores de uno y dos bits únicamente miran el comportamiento que ha ido teniendo un salto concreto. El código siguiente muestra el peor caso que puede aparecer en el predictor de 2 bits:

```
if (aa=2)
        aa=0;
if (bb=2)
        bb=0;
if (aa!=bb)
{
        …
}
```

En este fragmento de código el comportamiento del tercer salto depende del comportamiento de los dos saltos anteriores. Únicamente en el caso en que aa y bb sean iguales el tercer salto saltará. Los predictores que miran el comportamiento de los saltos anteriores reciben el nombre de "*Predictores Correlativos*" (Correlating Predictors) o bien "*Predictores de 2 niveles*" (Two-level Predictors). Para ver mejor el comportamiento de este tipo de predictores pondré un ejemplo:

```
if (d==0)
        d=1;
if (d==1)
        …
```

Para ver este ejemplo nos imaginaremos un bucle que va repitiendo el fragmento de código anterior y no se tendrán en cuenta ningún otro salto en la secuencia, ni siquiera el salto que hace que se repita el bucle. Supondremos también que en la ejecución d solo puede tomar los valores 0, 1 y 2. El comportamiento de los saltos respecto a estos los valores 0, 1 y 2 de d se muestran en la tabla siguiente.

Valor inicial	d==0?	Acción del primer if	Valor de d en el segundo if	d==1?	Acción del segundo if
0	Cierto	No salta	1	Cierto	No salta
1	Falso	Salta	1	Cierto	No salta
2	Falso	Salta	2	Falso	Salta

Tabla 2.1: Análisis de un predictor de 1 bit.

Un predictor de un bit iniciado en el estado de no saltar:

d=?	predicción primer if	Acción primer if	predicción siguiente primer if	predicción segundo if	Acción segundo if	predicción siguiente segundo if
2	No salta	Salta	Salta	No salta	Salta	Salta
0	Salta	No salta	No salta	Salta	No salta	No salta
2	No salta	Salta	Salta	No salta	Salta	Salta
0	Salta	No salta	No salta	salta	No salta	No salta

Tabla 2.2: Análisis del comportamiento de un predictor de 1 bit iniciado en el estado de no saltar.

Como se puede comprobar, fallan todas las predicciones.

Ahora vamos a ver que pasa si le colocamos un bit de correlación. La forma mas sencilla de ver este método es imaginando que tenemos dos bits separados, uno que controla si la ultima predicción fue salto y otro que mira si la ultima predicción fue de no saltar. En la tabla siguiente se muestra el comportamiento del predictor en sus diferentes casos:

Bits de predicción	Predicción si el ultimo salto no saltó	Predicción si el ultimo salto saltó
No salta / no salta	no salta	no salta
no salta / salta	no salta	salta
salta / no salta	salta	no salta
salta / salta	salta	salta

Tabla 2.3: Análisis de un predictor de 1 bit con 1 bit de correlación.

Vamos a ver como predice ahora los casos que hemos descrito anteriormente:

d=?	Predicción primer if	Acción primer if	Predicción siguiente primer if	Predicción segundo if	Acción segundo if	Predicción siguiente segundo if
2	**NS**/NS	S	S/NS	NS/**NS**	S	NS/S
0	S/**NS**	NS	S/NS	**NS**/S	NS	NS/S
2	**S**/NS	S	S/NS	NS/**S**	S	NS/S
0	S/**NS**	NS	S/NS	**NS**/S	NS	NS/S

Tabla 2.4: Análisis del comportamiento de un predictor de saltos de un bit con un bit de correlación.
Notación: Para abreviar, mostraremos salta como S y no salta como NS.

Ahora se puede comprobar la mejora que se ha obtenido. El único fallo de la tabla ha sido la primera predicción. Este predictor recibe el nombre de (1,1) predictor. Igual que en los casos anteriores, este predictor puede tener más de 1 bit de tamaño. El caso general del predictor de dos niveles seria un predictor (m,n). El predictor (m,n) mirara el comportamiento de los últimos m saltos para escoger 2m predoctores. Cada predictor será de n bits.

La ventaja del predictor de dos niveles respecto al predictor de dos bits es que con muy poca circuitería extra podemos aumentar el número de aciertos de nuestro predictor.

Un predictor de 2 bits solo utiliza información local para predecir el salto, esto hace que falle en predicciones importantes. Si al predictor de dos bits le añadimos información global su rendimiento se ve incrementado. El siguiente tipo de predoctores será aquel que incluya dos predoctores al mismo tiempo. El predictor a utilizar dependerá de un algoritmo previo que escogerá el más

adecuado al salto que se estudia. En la figura 9 se puede observar un ejemplo de un posible comportamiento de un algoritmo de dos bits encargado de escoger un predictor u otro.

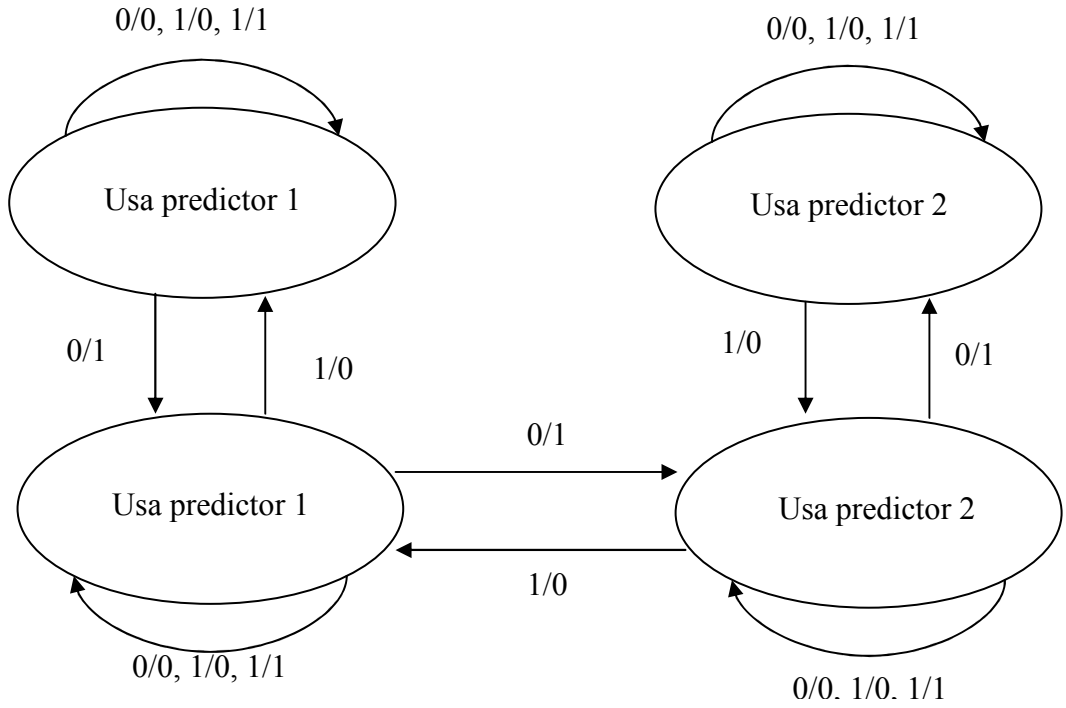


Figura 2.9: Esquema de funcionamiento del elector de predictor en un predictor de saltos híbrido.

El algoritmo consta de un contador que va siendo incrementado según se va acertando en la predicción. Y según se va fallando vamos decrementando.

2.5. Benchmarks

Un benchmark es un conjunto de aplicaciones encargadas de medir el rendimiento de un procesador. Lo importante en la ejecución de un benchmark no es el resultado obtenido por la propia aplicación, es el resultado que se obtiene al analizar el comportamiento del procesador mientras se esta ejecutando.

Como todo el mundo sabe, un ordenador se dedica a hacer cálculos matemáticos simples, sumas, restas, divisiones y productos. Un benchmark se dedica a realizar este tipo de operaciones millones de veces durante su ejecución. Ahora bien, existen dos grandes grupos en los números que entran en una maquina para realizar las operaciones: enteros y punto flotante. De modo que para analizar todos los cálculos posibles, los benchmarks quedan divididos en tres categorías:

- Enteros: Los números enteros son los números ordinales que todo el mundo conoce, negativos y positivos.
- Punto flotante: Los números en punto flotante son aquellos que disponen de decimales.
- Transacciones on-line y off-line sobre las bases de datos.

Dentro de estos grupos también podemos subdividir los benchmark según la naturaleza del programa, de esta manera nos quedan los subgrupos siguientes:

- Programas reales: (compiladores, procesadores de texto, …) con estas aplicaciones se obtienen resultados mas precisos y permiten diferentes formas de ejecución.
- Núcleos (kernels): Son trozos de aplicaciones reales. Están preparados par analizar características de rendimiento especificas de una determinada maquina. (Limpack, Livermore Loops, …)
- Patrones de conjunto (benchmarks suits): Serie de aplicaciones encargadas de medir los diferentes modos de funcionamiento de una maquina. (SPEC y TPC).
- Patrones reducidos (toy benchmarks): Programas de pocas líneas de código (entre 10 y 100) y de resultado conocido. Son fáciles de ejecutar en cualquier maquina. (Quicksort, …).
- Patrones sintéticos (synthetic benchamrks): código artificial que no pertenece a ningún programa de usuario y que se utiliza para determinar perfiles de ejecución. (Whetstone, Dhrystone).

Nuestro estudio trata los benchmark SPEC, en concreto los utilizados para el estudio son los SPEC95 mientras que los realizados para la comprobación final fueron los SPEC2000. SPEC (System Performance and Evaluation Cooperative) es una sociedad sin ánimo de lucro que tiene como misión establecer, mantener y distribuir un conjunto estandarizado de benchmarks que se pueden aplicar a las últimas generaciones de procesadores. Se han sucedido ya 4 generaciones de benchmarks: SPEC89, SPEC92, SPEC95 y SPEC2000. A continuación comentaremos la más reciente.

SPEC CPU2000 Los diseña el OSG (Open System Group) de SPEC que es el encargado de los benchmarks de componentes y sistemas. En este apartado se encuentran los SPECint2000 y SPECfp2000, representativos de las aplicaciones enteras y en punto flotante. Dentro de SPEC existen otros dos grupos HPG (High Performance Group) y GPCG (Graphics Performance Caracterization Group).

La maquina de referencia de los SPEC CPU2000 es la UltraSparc10 con un procesador UltraSPARC Iii, a 300 Mhz y 256 Mb de memoria. Esto significa que todos los resultados se calculan como en función de esta maquina, que por definición tiene SPECint2000=100 y SPECfp2000=100.

Veamos ahora cuales son las aplicaciones que componen cada grupo:

SPECint2000:

- Gzip: programa de compresión de datos que utiliza el algoritmo de Lempel-Ziv (LZTT).
- Vpr: (versátil place and route) implementa los procesos de ubicación y conexionado de un circuito digital sobre bloques lógicos de FPGA's. Pertenece a la clase de programas de optimización combinatoria. Utiliza simulated annealing como procedimiento de optimización.

- Gcc: compilador de C que genera código para el procesador Motorota 88100.
- Mcf: Programa de planificación temporal (scheduling) de transporte público que utiliza el código MFC del método simplex de red (versión especializada en el algoritmo simplex de programación lineal).
- Crafty: Programa de juego de ajedrez por computador con un numero significativo de operaciones enteras y lógicas tales como AND, OR, XOR y desplazamientos.
- Parser: Analizador sintáctico de ingles basado en gramáticas de enlace (link grammar) con un diccionario de mas de 60.000 palabras.
- Eon: Trazador de rayos (ray tracer) que transmite un numero de líneas 3D (rayos) sobre un modelo poligonal 3D, calcula la intersección entre las líneas y los polígonos, y genera nuevas líneas para calcular la luz incidente en los puntos de intersección. El resultado final es una imagen vista por cámara.
- Perlbmk: Versión del lenguaje de script Perl en el que se han eliminado las características especificas del SO.
- Gap: (Groups Algorithm and Programming) Implementa un lenguaje diseñado para computar en grupos.
- Vortex: Procede de un OODBMS (Sistema de Gestion de Bases de Datos Orientados a Objetos) que se ha adaptado para conformarlo a las exigencias de SPEC2000.
- Bzip2: Compresor basado en la versión 0.1 de bzipb.
- Twolf: Paquete de ubicación y conexionado en el proceso de diseño de circuitos integrados basados en celdas standard. Utiliza un algoritmo de simulated annealing.

SPECfp2000:

- Wupwise: Cromodinámica quántica.

- Swim: modelado del agua superficial.
- Mgrid: resolutor multi-malla, campo potencial 3D.
- Applu: ecuaciones diferenciales parciales elípticas/parabólicas.
- Mesa: librería gráfica 3D
- Galgle: Dinámica de fluidos.
- Art: reconocimiento de imágenes/redes neuronales.
- Equake: Simulación de propagación de ondas sísmicas.
- Facerec: Procesamiento de imagen, reconocimiento de rostro.
- Ammp: química computacional.
- Lucas: Teoría de números/prueba de números primos.
- Fma3d: simulación de choques por elementos finitos.
- Sixtrack: diseño de aceleradores de física nuclear de alta energía.
- Apsi: meteorología, distribución de contaminantes.

Uno de los datos que se pueden obtener de la ejecución de un benchmark es la velocidad de ejecución del procesador. La velocidad va marcada por el número de instrucciones que es capaz de ejecutar el procesador por unidad de tiempo. Generalmente la unidad en la que se mide la velocidad del procesador son los MIPS (Millones de Instrucciones por Segundo) en el caso de los benchmak de enteros y MFLOPS (Millones de Instrucciones en Punto Flotante por Segundo) en el caso de los benchmark de punto flotante.

Para poder sacar los resultados en MIPS i/o MFLOPS, las aplicaciones a ejecutar deben disponer de millones de instrucciones para ser ejecutadas. El caso es que cada aplicación puede llegar a tener más de 1 Billón de instrucciones. A un billón le llamaremos según el sistema de medida americano 1 billón = 1.000 millones. En concreto las aplicaciones que nosotros hemos empleado para realizar el estudio tienen:

li.eio	->	4,5 billones de instrucciones
go.eio	->	5,4 billones de instrucciones
m88k.eio	->	29,7 billones de instrucciones

Los resultados obtenidos para medir la velocidad son los más rápidos, dado que no tenemos que analizar a fondo el funcionamiento del procesador.

2.6. SimpleSim

SimpleSim es un simulador de un procesador Alpha. El simulador se encarga de ejecutar un fichero llamado EIO ya creado, a través de un método que se vera mas adelante, como si se hiciera en un procesador Alpha y con las característica que se le pasa en los parámetros que también se explicaran con posterioridad.

El simulador SimpleSim es una aplicación de libre distribución y código abierto. El código fuente esta hecho en C. El código fuente se puede descargar de la Web referenciada en la bibliografía. En este caso, seguiremos unas instrucciones para compilar el código fuente en una maquina Linux corriendo con una distribución Debian 3.0. La versión de SimpleSim que se utilizará para la realización de este proyecto es la 3.0. Una vez se dispone del código fuente, se debe editar el fichero "*Makefile*" para indicarle sobre que maquina se va a compilar el simulador. Posteriormente se debe compilar el simulador que se desea utilizar, en este caso el simulador es el de una maquina Alpha, por lo que deberemos escribir "*make config*" en la línea de comandos. Ahora ya se dispone de la herramienta que nos permitirá realizar las simulaciones de la maquina Alpha.

Una vez compilada la aplicación se puede apreciar una serie de ficheros ejecutables, cada uno para realizar una tarea distinta. Los ejecutables se muestran en la lista siguiente:

- sim-bpred
- sim-cache
- sim-eio
- sim-fast

- sim-outorder
- sim-profile
- sim-safe

De todas estas aplicaciones se utilizaran las relacionadas con el predictor de saltos, que serán: sim-bpred, sim-eio y sim-fast.

La aplicación *sim-eio* nos servirá para generar lo que llamaremos checkpoint y para generar los ficheros EIO que posteriormente servirán para ejecutar la simulación que mas interese, en nuestro caso la del predictor de saltos.

Un checkpoint es una marca que se hace en un fichero EIO para cuando se realice la simulación se ignoren todas las instrucciones hasta llegar a la marca. Esta ejecución nos servirá para poder decidir posteriormente que partes de la simulación no son significativas para el resultado y así ahorrar tiempo analizándolas.

Un fichero EIO es un fichero binario que contiene los datos de ejecución de una aplicación determinada así como los parámetros necesarios para dicha ejecución. El hecho de guardarlo como fichero EIO nos permitirá poder lanzar la ejecución en un simulador como SimpleSim.

Los parámetros que se pueden introducir a la aplicación son:

- config: carga la configuración de un fichero
- h: muestra un mensaje de ayuda
- v: verbose operation
- d: active el mensaje de debugar
- i: modo interactivo
- seed: generador de numero aleatorio
- q: inicializa y/o termina
- chkpt: carga un fichero EIO de checkpoint
- redir:sim: redirecciona la salida del simulador a un fichero
- redir:prog: redirecciona la salida del programa a un fichero
- nice: programador de prioridad del simulador

- max:inst: numero máximo de instrucciones a ejecutar
- fastfwd numero de instrucciones que se salta antes de empezar la simulación.
- trace: guarda el resultado en un fichero de trazas
- perdump: genera un checkpoint cada N instrucciones
- dump: especifica un fichero de checkpoint y un rango

Con esta información, se puede fácilmente deducir que para generar un fichero EIO bastaría con ejecutar

"*sim-eio fichero_origen [parametros] fichero.eio*"

De este modo generaríamos un fichero EIO a través de un fichero origen "fichero_origen" pasándole los parámetros necesarios para su ejecución.

Para generar un fichero de checkpoint, la línea a escribir seria:

"*sim-eio –dump file.chk 10000000 file.eio*"

En este caso se generaría un fichero de checkpoint del fichero "file.eio" llamado "file.chk" el cual tendría una marca en la instrucción 10.000.000. De este modo se puede cargar el fichero en otra aplicación de las vistas anteriormente y ejecutar de forma rápida (sin analizar el procesador) las primeras 10 millones de instrucciones, y a partir de ese punto el simulador empezaría a analizar las siguientes instrucciones ejecutadas.

La aplicación sim-bpred es la que simula el predictor de saltos [2.5]. A esta aplicación se le tienen que pasar los parámetros necesarios para poder simular el predictor que se desea emplear. Los parámetros de que se dispone son:

- config: carga la configuración de un fichero
- dumpconfig: guarda la configuración en un fichero
- h: muestra un mensaje de ayuda
- v: verbose operation
- d: active el mensaje de debugar
- i: modo interactivo

- seed: generador de numero aleatorio
- q: inicializa y/o termina
- chkpt: carga un fichero EIO de checkpoint
- redir:sim: redirecciona la salida del simulador a un fichero
- redir:prog: redirecciona la salida del programa a un fichero
- nice: programador de prioridad del simulador
- max:inst: numero máximo de instrucciones a ejecutar
- bpred tipo de predictor a utilizar: {nottaken|taken|bimod|2lev|comb}
- bpred:bimod: configuración del predictor bimodal
- bpred:2lev: configuración del predictor de 2 niveles
- bpred:comb configuración del predictor híbrido
- bpred:ras tamaño de la pila para la dirección de retorno
- bpred:btb tamaño para la tabla BTB

Con estos datos es fácil deducir que una ejecución normal de un predictor de saltos de tipo bimodal cuya tabla tiene un tamaño 1024 seria escribir en la línea de comandos:

"*sim-bpred –bpred bimod –bpred:bimod 1024 file.eio*"

Otra posible ejecución de la aplicación sim-bpred puede ser:

"*sim-bpred –bpred bimod –bpred:bimod 1024 –redir:sim /tmp/out.txt file.eio*"

Esta línea únicamente se diferencia de la anterior en que los resultados del simulador se redirecciona al fichero *out.txt* en vez de salir por pantalla. Este método servirá posteriormente para poder analizar con más detenimiento los datos obtenidos en la simulación y poder pasarlos a otros formatos para poder tratarlos adecuadamente.

Por ultimo, la aplicación sim-fast se encarga de ejecutar un fichero eio sin analizar en profundidad el procesador, de modo que el único resultado que se obtendrá es el de tiempo de ejecución e instrucciones ejecutadas. Esto servirá para poder medir la velocidad a

la que podemos ejecutar los trozos no analizados en el posterior análisis. Los parámetros que ofrece esta aplicación son:

- config: carga la configuración de un fichero
- dumpconfig: guarda la configuración en un fichero
- h: muestra un mensaje de ayuda
- v: verbose operation
- d: active el mensaje de debugar
- i: modo interactivo
- seed: generador de numero aleatorio
- q: inicializa y/o termina
- chkpt: carga un fichero EIO de checkpoint
- redir:sim: redirecciona la salida del simulador a un fichero
- redir:prog: redirecciona la salida del programa a un fichero
- nice: programador de prioridad del simulador

Una ejecución simple de esta aplicación podría ser:

"*sim-fast –redir:sim /tmp/out.txt file.eio*"

Esta ejecución únicamente simularía la ejecución del fichero 'file.eio' y el resultado de la simulación lo guardaría en el fichero 'out.txt'.

3. Análisis del problema y propuesta

La ejecución o simulación de un benchmark en una máquina es un proceso que puede tardar días en terminar. La finalidad de este proyecto es reducir ese tiempo teniendo unos resultados de la simulación lo más parecidos posible a si simuláramos de forma normal. Para poder reducir el tiempo de simulación, una de las alternativas que se nos presenta es la de suprimir de la simulación intervalos de la aplicación que se consideren menos significativos para el resultado final.

La simulación completa de un benchmark consiste en hacer que se ejecute todo el benchmark (desde la primera instrucción hasta la última) con unos determinados parámetros de configuración de un procesador ficticio. La simulación completa representa el caso que utilizaremos de referencia para reducir el tiempo de simulación y obtener un resultado lo más próximo a éste. Por otra parte, la simulación incompleta parte de la base de no simular toda la aplicación. La simulación incompleta simulará únicamente un cierto número de intervalos.

Un intervalo de simulación se define como un cierto número de instrucciones que pertenecen a la simulación de una aplicación. El intervalo se mantendrá fijo durante toda la simulación. Esto quiere decir, que si por ejemplo se escoge un intervalo de 100.000 instrucciones, el primer intervalo irá de la primera instrucción hasta la instrucción 100.000, el segundo empezará en la instrucción 100.001 y terminará en la 200.000, el tercero empezará en la 200.001 y terminará en la 300.000, y así sucesivamente.

El hecho de simular por intervalos e incluso el hecho de suprimir intervalos durante la simulación para hacer que tarde menos tiempo hace que se cometan imprecisiones en el resultado final. El error absoluto es la diferencia del resultado de la simulación en un determinado intervalo entre la simulación completa y la simulación

incompleta. En este estudio el resultado de las simulaciones es el número de fallos que realiza un predictor en cada intervalo.

$$Error\ Absoluto = (resultado\ simulación\ completa) - (resultado\ simulación\ incompleta)$$

El error relativo será el porcentaje obtenido del error absoluto respecto de la simulación incompleta.

$$Error\ Relativo = \frac{Error\ absoluto}{(Resultado\ Simulación\ incompleta)}\ x\ 100$$

Hay dos formas de hacer bajar el tiempo de simulación de una aplicación, mediante un único procesador o mediante varios procesadores. Primero estudiaremos el caso de un solo procesador.

En la simulación incompleta podemos distinguir dos tipos de simulaciones. Veamos los casos:

3.1. Incremental

La simulación incompleta incremental parte de empezar a simular un intervalo cualquiera, por ejemplo el intervalo central. El valor que obtengamos de esta simulación será utilizado para compararlo con el resultado de la siguiente simulación.

En el segundo paso de simulación el resultado se obtendrá del intervalo que se simuló en el paso anterior y además los intervalos que queden en el medio de cada una de las partes de aplicación que quedan por simular, véase mejor la referencia en la figura 4.1.

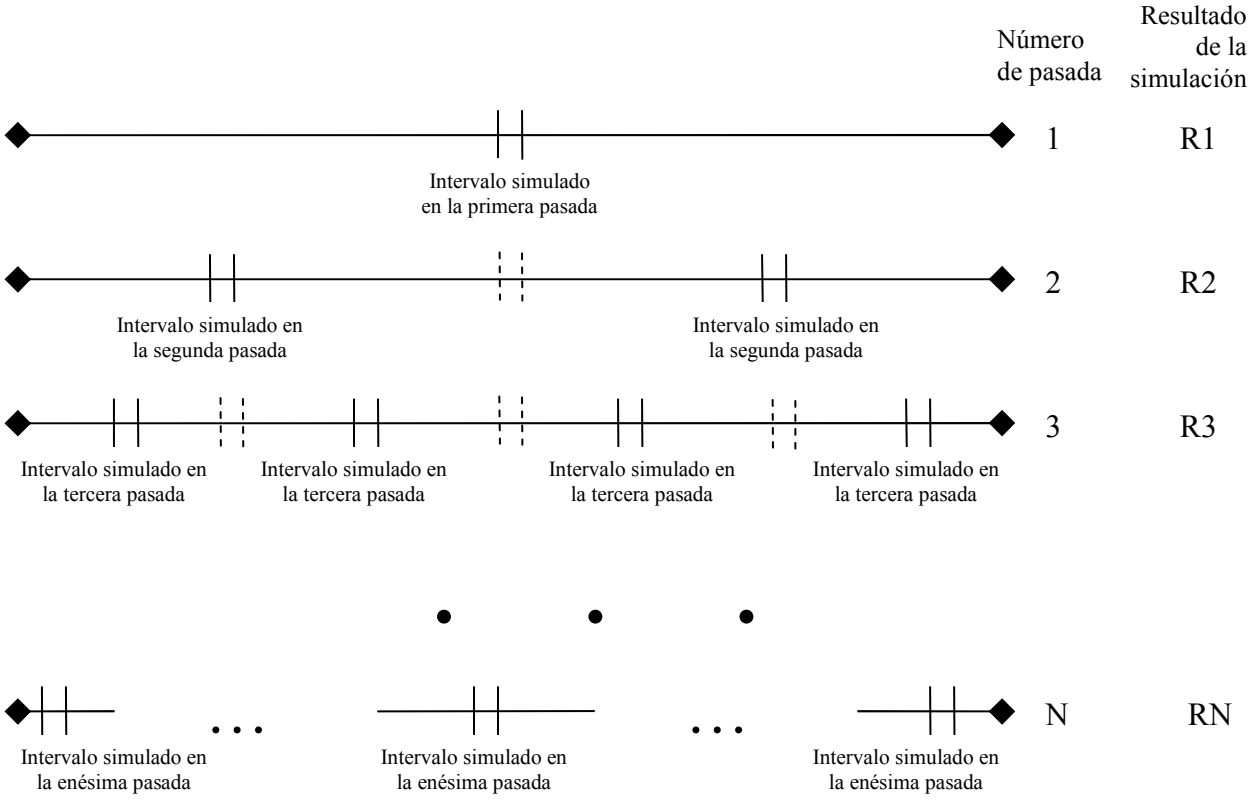


Figura 4.1: Muestra las pasadas que se realizan y que intervalos deben escogerse en cada una de ellas.

La idea general del algoritmo sería ir simulando cada vez más intervalos teniendo en cuenta los que ya se han simulado y comparar el resultado obtenido con el anterior.

Para empezar el algoritmo se recomienda comenzar por un intervalo central. El escoger el intervalo del centro nos permite ir escogiendo cada vez el intervalo central de cada parte en la que se divide el programa, así se evita escoger siempre los intervalos pertenecientes a un mismo fragmento de programa y por tanto la variedad del resultado permite obtener un resultado final mas parecido al de la simulación completa. Veamos paso a paso como seria la realización del algoritmo:

$$Error\ Relativo = \frac{R2 - R1}{R2}$$

Del resultado obtenido en la segunda pasada (la segunda pasada tiene 3 intervalos, 1 obtenido en la primera pasada más 2 obtenidos en la segunda pasada) de la simulación se debería calcular el error relativo respecto al resultado obtenido en la pasada anterior (en este caso el de la primera pasada que solo tiene un intervalo).

Siguiendo este procedimiento se deberían seguir simulando más pasos hasta obtener un error relativo del paso i+1 respecto al paso i inferior a un cierto valor ε dado antes de empezar la simulación incompleta incremental.

$$Error\ Relativo = \frac{RN - R(N\text{-}1)}{RN} < \varepsilon$$

El parar en el paso i+1 de la simulación incompleta no asegura que el error respecto a la simulación completa sea inferior a ε pero la probabilidad que sea próximo a ε es bastante alta.

3.2. Análisis previo de intervalos significativos

En este caso se lanza una simulación completa secuencial para una configuración, pongamos por ejemplo el caso de un predictor bimodal de un tamaño determinado. Una vez se tiene el comportamiento de la aplicación se hace simulación incompleta para el resto de tamaños del mismo tipo de predictor, en este caso bimodal. Si se desea cambiar de predictor, por ejemplo estudiar el predictor de dos niveles, se debería volver a realizar una simulación completa para un tamaño "estándar" y comprobar así su comportamiento para luego realizar simulaciones incompletas. Para poder hacer un estudio más exhaustivo de la simulación se pueden utilizar herramientas preparadas para el hecho. Una de estas herramientas se llama "simpoint". El "Simpoint" realiza una simulación funcional que quiere decir que no simula cache, predictor

de saltos ni unidades de ejecución, con lo que la simulación es mucho más rápida y se puede obtener un resultado del comportamiento de la aplicación en mucho menos tiempo.

3.3. Simulación paralela

Hasta ahora se ha explicado como se podría mejorar el tiempo de simulación de los benchmark con un solo ordenador. La simulación paralela consiste en simular la aplicación en varios procesadores al mismo tiempo. Esto permitirá enviar los intervalos calculados a cada uno de los procesadores de que se dispone y así obtener un resultado en menos tiempo que si se hiciera con un solo procesador.

La simulación paralela completa simulará toda la aplicación dividiéndola en intervalos y enviando los intervalos a cada uno de los procesadores. El resultado que se obtenga de esta simulación no será el mismo que si se hiciera en un único procesador. El comenzar un intervalo en un procesador sin mantener la historia de lo que ha estado haciendo el predictor en las instrucciones precedentes hará que empiece fallando las primeras instrucciones.

La simulación paralela incompleta es similar a la simulación incompleta antes comentada solo que los intervalos en vez de simularse todos en la misma maquina se envían a simular cada uno a una maquina diferente, de este modo se obtiene el resultado en menos tiempo. El poder simular intervalos diferentes en procesadores diferentes nos abre un abanico de posibilidades que con un solo procesador no tenemos.

Se pude utilizar cada procesador para simular con una configuración diferente aplicando el algoritmo antes descrito. Con este método se puede obtener el resultado de varias configuraciones al mismo tiempo.

Otra posibilidad es dividir la aplicación en tantos trozos como procesadores, simulando cada uno con la misma configuración y que

cada uno aplique el algoritmo. La diferencia entre este y el anterior es que se puede reducir el ε y obtener el resultado con mayor precisión dado que se habrá simulado más intervalos.

3.4. Checkpoint

Una vez explicado todo lo expuesto se puede pasar a mostrar las ventajas e inconvenientes de un checkpoint. Un checkpoint es una marca en un punto determinado de la simulación. Esta marca sirve para reanudar la simulación de la aplicación desde ese punto manteniendo intacto el estado de la máquina. Por el contrario el predictor de saltos pierde la historia al intentar reanudar en el punto marcado con el checkpoint.

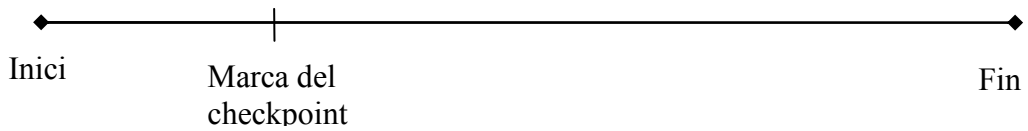


Figura 4.3: Simulación con una marca de checkpoint en un punto cualquiera.

Para entender mejor que se guarda con un checkpoint imaginemos por un momento la ejecución de una aplicación. Los registros del procesador van variando su contenido conforme van avanzando las instrucciones, cada uno de los registros contiene los datos, quizá necesarios, de instrucciones posteriores. El predictor de saltos se va llenando con la información de los saltos realizados y no realizados. Y la máquina va adquiriendo un estado determinado en cada instrucción. El estado que tiene la máquina cuando llega al punto marcado se tiene que guardar para poder reanudar desde ahí. Lo que se tiene que guardar es información trascendental, por ejemplo el resultado de una comparación hecho justo en la instrucción precedente. La información no necesaria se puede suprimir para que el fichero de checkpoint que guarda el estado de la

máquina en ese momento no sea demasiado grande, por lo que la información de la historia del predictor de saltos se puede omitir, así como la memoria cache. El perder la información de la historia del predictor de saltos hará que la probabilidad que el predictor de saltos falle sea más alta que si la simulación no se hubiera detenido al pasar por el checkpoint, por lo que consideraremos esto como un inconveniente a la hora de generar checkpoints.

Una ventaja que ofrece el tener checkpoints de una simulación es el no tener que simular la parte que le precede, esto servirá para cuando se tengan que hacer más checkpoints en la simulación paralela.

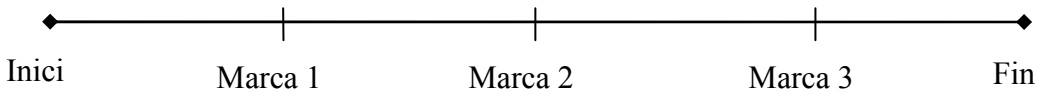


Figura 4.4: Simulación con 3 marcas de checkpoint

Supongamos por ejemplo que tenemos la simulación anterior en la que hemos generado tres checkpoints. El error obtenido en esta simulación es superior al esperado, por lo que se decide generar cuatro checkpoints más. Cada checkpoint se generará en la mitad de cada fragmento de simulación que hay entre marcas. Se empieza a simular desde el inicio, al llegar hasta la mitad de entre inicio y marca 1 se genera un checkpoint.

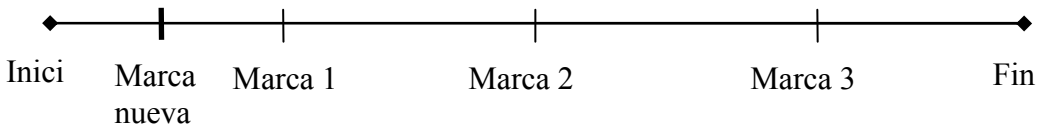


Figura 4.5: Simulación idéntica a la de la figura 4 pero con un nuevo checkpoint entre inicio y marca 1.

Una vez generado el checkpoint, nombrado como "marca nueva" en la figura 4.5 pasamos a generar el checkpoint que va entre Marca

1 y Marca 2. En vez de continuar simulando, aprovechamos el checkpoint generado en Marca 1 y empezamos desde ahí. De este modo se puede ahorrar el simular la parte marcada como discontinua en la figura 4.6. El fragmento marcado con la doble flecha es el fragmento simulado para generar el checkpoint. Para generar el checkpoint que iría entre Marca 2 y Marca 3 se debería seguir el mismo proceso, empezar en Marca 2 y simular hasta la mitad para generar el checkpoint.

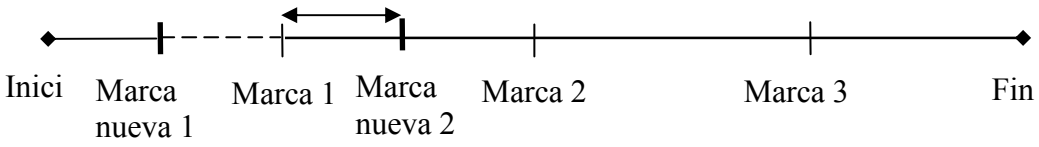


Figura 4.6: Simulación idéntica a la de la figura 5 pero con un nuevo checkpoint entre marca 1 y marca 2.

Como se puede deducir de forma fácil, la generación de los nuevos checkpoints tardara la mitad del tiempo que si se hiciera una ejecución completa para generarlos dado que aproximadamente la mitad de la simulación se puede despreciar.

Otro inconveniente que pueden ofrecer los checkpoints es el tamaño que ocupan. Un checkpoint puede llegar a alcanzar los 100 Mb de tamaño, por lo que el generar muchos haría necesario tener un disco de gran tamaño para poder albergarlos a todos. El tamaño también se puede considerar un inconveniente si se intenta pasar a través de una red los checkpoints hacia otras maquinas que los requieran en una simulación incremental paralela ya que quizá seria mas efectivo que cada maquina generara sus propios checkpoints

3.5. Nuestra propuesta

Una vez presentados todos los aspectos técnicos necesarios para describir este proyecto pasamos a analizar el problema de forma práctica.

Como ya se ha comentado anteriormente la idea es reducir el tiempo de simulación de un benchmark y al mismo tiempo obtener un resultado lo mas parecido posible al que se obtendría si se simulara de forma normal. Para poder reducir ese tiempo se ha optado por empezar simulando pocos intervalos e ir aumentando el número de intervalos simulados gradualmente hasta que el error respecto a la simulación anterior sea suficientemente pequeño.

Antes que nada explicaremos que pasos se deberían seguir para la realización de una simulación completa. Los ficheros con los que se realizan las pruebas son ficheros binarios para una máquina concreta. A los ficheros binarios se les debe pasar una serie de parámetros determinados, que no entraremos a analizar. Posteriormente se debe pasar el fichero binario por un simulador para obtener un fichero con un formato especial para ser simulado por el simulador, este fichero recibirá el nombre de "fichero eio" dado que la extensión que tendrá será eio. El siguiente paso será el de realizar la simulación completa de la aplicación para cada una de las configuraciones que se desea probar.

Todo este proceso, aunque parece sencillo es bastante costoso en tiempo de cómputo. Ahora explicaremos una posibilidad para poder reducir el tiempo de simulación. Para acelerar el proceso de creación de checkpoints y del fichero eio, al mismo tiempo que generamos el fichero eio podemos generar un número determinado de checkpoints dependiendo del tamaño que ocupen. El siguiente paso es simular los intervalos en los checkpoints calculados utilizando el algoritmo antes descrito del método incremental. Si resultara que el número de intervalos que se ha simulado con los checkpoints calculados no fuera suficiente, es decir, que el error del último paso

respecto del anterior es superior al error definido, se debería utilizar la aplicación sim-eio para generar más checkpoints. Una vez generados se van simulando continuando con los incrementos previstos en el algoritmo. Los checkpoints utilizados en la simulación de la primera configuración se pueden "reciclar" y reutilizar para el resto de configuraciones.

4. Fases del proyecto

En este apartado se hablará de cómo se va a plantear el proyecto, las estrategias a seguir y la metodología empleada. Se comentaran los ficheros que se decidieron coger para realizar el estudio. También se comentarán unas primeras pruebas realizadas con dichos ficheros y sus resultados. Los resultados de estas primeras pruebas pueden dar una idea de hacia donde seria mejor encaminar el análisis. Al final del capitulo se muestran una serie de problemas encontrados en las simulaciones, y análisis de las pruebas y las soluciones que se han propuesto.

4.1. Ficheros utilizados

Durante la realización del estudio se han utilizado tres ficheros: go.eio, li.eio y m88k.eio pertenecientes a los Benchmark SPEC95. Estos ficheros han servido para estudiar el comportamiento de sus ejecuciones y sacar unos resultados que han servido para llegar a unas conclusiones que posteriormente se han ratificado con otras aplicaciones. Para la comprobación del buen funcionamiento del método se han empleado cuatro aplicaciones pertenecientes al SPEC00. Las aplicaciones son: TWOLF, GCC200, PARSER y MCF.

4.2. Empezando el estudio

El estudio del comportamiento de las aplicaciones go.eio, li.eio y m88k.eio se ha empezado realizando unas simulaciones cortas respecto al tamaño de la aplicación, siempre hablando en número de instrucciones. Esto servirá posteriormente para poder encaminar el estudio hacia una rama u otra en función del resultado obtenido.

Se empezó ejecutando una serie de simulaciones de 1 millón de instrucciones para comprobar el resultado que se obtenía. El

resultado apreciado en tiempo era siempre de 1 segundo, por lo que se pasó a ejecutar mas instrucciones para comprobar con algo más de exactitud el tiempo que se tardaba en ejecutar las instrucciones. Para que el resultado fuera más significativo se procedió a multiplicar por 10 el número de instrucciones ejecutadas, de modo que se ejecutaban intervalos de 10 millones de instrucciones. El resultado obtenido fue algo más satisfactorio, los tiempos rondaban entre 7 y 8 segundos.

Pese a haber obtenido resultados mas ajustados en numero de instrucciones por unidad de tiempo, se decidió lanzar mas simulaciones en intervalos mas grandes. El intervalo escogido esta vez fueron 100 millones de instrucciones. Con estas ejecuciones, los resultados que se obtuvieron daban unos tiempos entre 72 y 76 segundos. De este modo podemos deducir que el simulador era capaz de simular entre 1,31 millones de instrucciones y 1,38 millones de instrucciones por segundo.

Estas pruebas nos servirán mas adelante para conocer el comportamiento del programa en una ejecución que llamaremos paralela. La ejecución en paralelo consiste en ir simulando la ejecución en intervalos marcados de instrucciones. Cada intervalo se empieza como si se empezara la ejecución desde cero. De esta forma se puede estudiar fácilmente que pasa al eliminar ciertos trozos de la aplicación.

Una vez estudiados los tiempos de simulación de las aplicaciones se procedió a estudiar la variación que habría al cambiar la configuración del predictor de saltos.

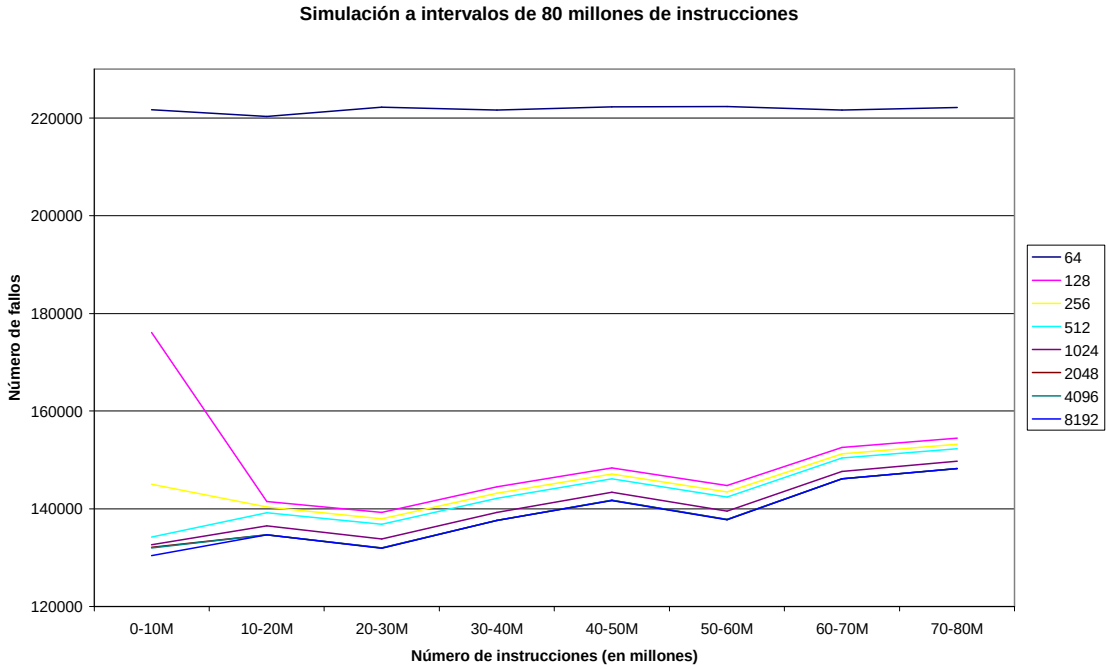


Figura 4.1: Gráfico que muestra el número de fallos por intervalo simulado en una simulación continúa de la aplicación li.eio con distintos tamaños de la tabla de historia.

El grafico de la figura 4.1 muestra el número de fallos que se producen en cada intervalo de 10 millones de instrucciones desde el principio de la aplicación hasta llegar a un tamaño máximo de 80 millones de instrucciones. Según muestra la leyenda, cada línea del grafico corresponde a un tamaño diferente del predictor de saltos en su configuración bimodal.

Los resultados de la simulación fueron obtenidos la simular de forma secuencial los 80 millones de instrucciones y obteniendo los resultados al llegar al final de cada intervalo. Para lograr esto se tuvo que ejecutar la aplicación tantas veces como intervalos se querían analizar. El resultado de la primera simulación pertenecía al intervalo 0-10 millones de instrucciones. En la segunda simulación se simulan de 0 a 20 millones de instrucciones, de ahí se resta el resultado obtenido en la simulación anterior (de 0 a 10 millones de instrucciones) y se obtiene el resultado deseado, el numero de fallos que pertenecen al intervalo de 10 a 20 millones de instrucciones.

Siguiendo este método se puede obtener fácilmente el resto de intervalos.

El siguiente paso del estudio previo fue comprobar las diferencias de simular de forma continua las aplicaciones o bien lanzar simulaciones paralelas. Las simulaciones paralelas son aquellas que simulan un intervalo determinado pero sin mantener la historia del predictor, es decir, todos los datos que el predictor ha podido ir guardando durante la simulación hasta llegar al intervalo a analizar se despreciará. Para poder obtener este resultado se debe primero crear un checkpoint (comentado anteriormente) en el intervalo deseado mediante la aplicación sim-eio. Una vez creado el checkpoint se simula mediante sim-bpred. Lo que hará sim-bpred es simular sin guardar información del predictor hasta llegar al punto marcado, y desde ese punto hasta el final del intervalo se guardara la información del predictor de saltos. El resultado para las configuraciones e intervalos anteriores se puede apreciar en la siguiente grafica (figura 4.2).

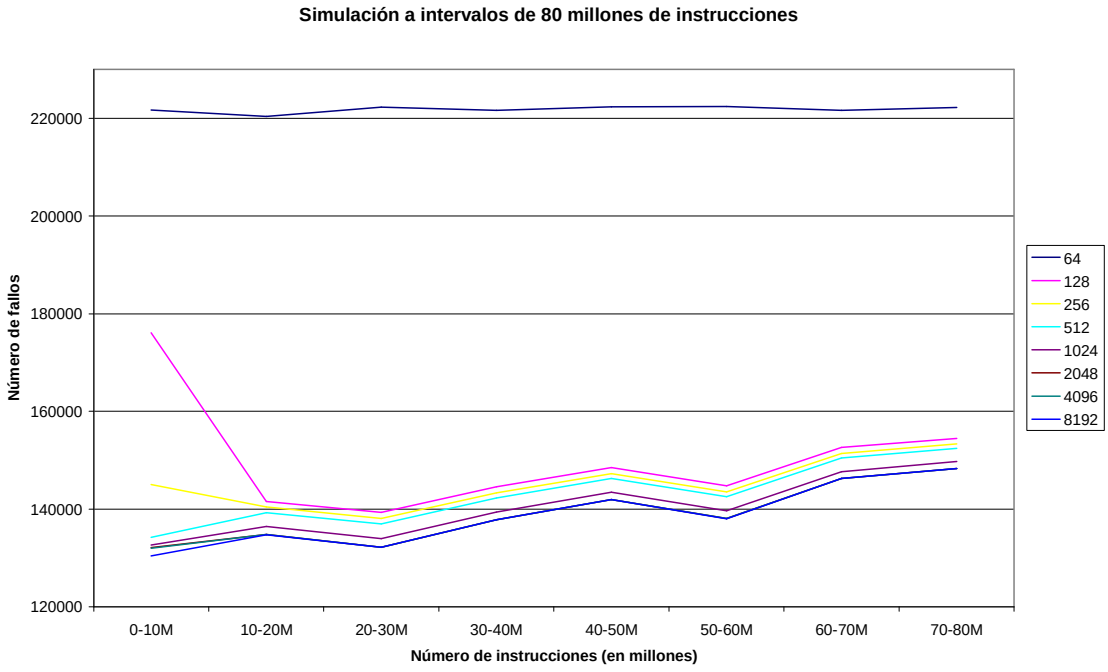


Figura 4.2: Gráfico que muestra el número de fallos por intervalo simulado en una simulación paralela de la aplicación li.eio con distintos tamaños de la tabla de historia.

Como se puede observar en la figura 4.2, el número de fallos que se obtiene al simular en modo paralelo es prácticamente igual al número de fallos obtenidos anteriormente en la simulación en serie. Este dato será de gran relevancia para cuando se decida suprimir un intervalo de la simulación de la aplicación, dado que, como se aprecia en el grafico, el número de fallos que pueden salir de más al tener que volver a iniciar la tabla del predictor no es significativo y por tanto se pude despreciar.

4.3. Elección de la configuración del predictor

Lo primero que se va a estudiar son las configuraciones que se emplearán durante el estudio. En nuestro caso, dado que se han probado diferentes tamaños de una misma configuración, se continuará el estudio con esa misma configuración utilizando únicamente dos de esos tamaños. Se escogen dos dado que las diferencias entre las distintas configuraciones no se apreciaban diferencia alguna en comportamiento de la aplicación. Únicamente se apreciaban diferencias en el número de fallos y también porque las simulaciones son muy largas y el hecho de coger más haría que el estudio se alargara de forma innecesaria. Se escogen los dos tamaños más representativos, el máximo y el mínimo, el resto de tamaños se puede suponer que quedan entre los valores que se obtendrán. Entonces durante el estudio se utilizaran simulaciones de predictor Bimod de 64 y Bimod de 8192.

4.4. Tiempo de Ejecución

Durante el siguiente apartado expondremos los tiempos que tarda el simulador en realizar los intervalos. De esta forma se pueden hacer una idea de lo que tardaría en tiempo real en realizar todas las instrucciones. Recordaremos también que no solo es simular todas las

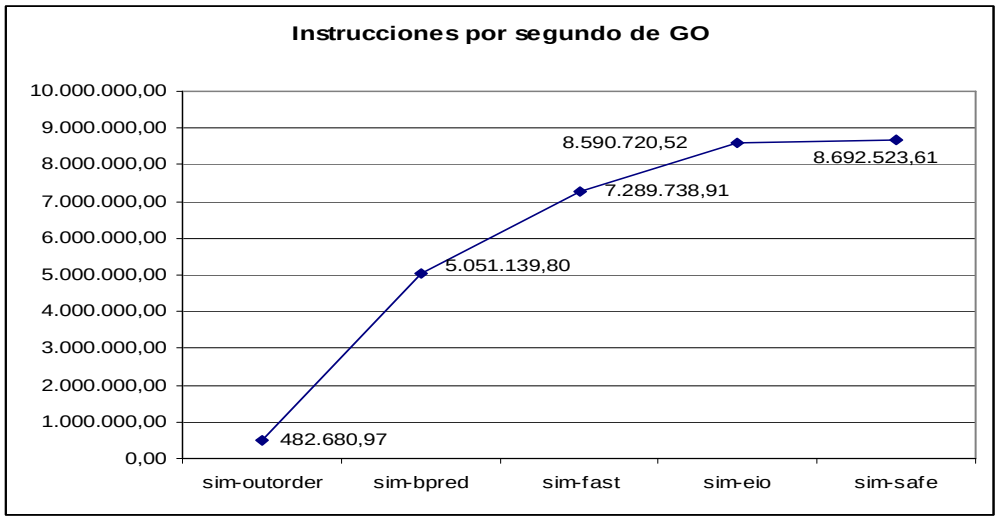
instrucciones sino que se deben simular con diferentes configuraciones.

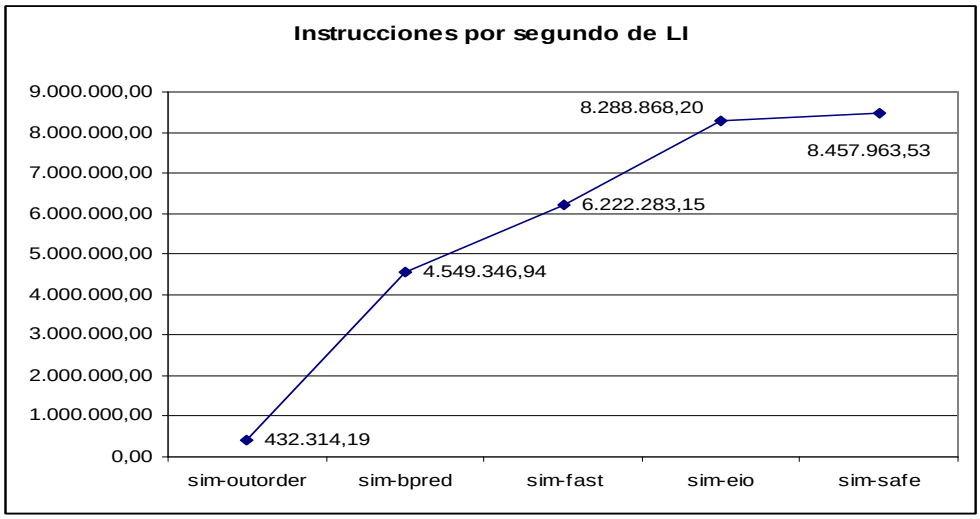
Los tiempos obtenidos para estos resultados han sido con los tres programas que llevamos analizando desde que se ha empezado este estudio: go, li y m88k. Para go hay exactamente 4.089.397.734 instrucciones, para li 5.526.619.447 y para m88k 29.965.659.025 instrucciones. Estos programas son muy pequeños si los comparamos con otros programas pertenecientes a diferentes benchmarks que llegan a superar los 100.000 millones de instrucciones.

Para analizar el tiempo que tarda el simulador en ejecutar las aplicaciones se han analizado simulaciones completas de diferentes métodos de simulación: sim-eio, sim-fast, sim-outorder, sim-safe, sim-bpred. Cada una de las simulaciones hace una función en concreto. Sim-eio es la parte del simulador que permite generar los ficheros EIO que después se pueden simular con el resto de simuladores. Sim-fast es el programa que simula analizando únicamente pequeños datos funcionales como valor de registros. Sim-safe simula igual que sim-fast pero añadiendo unas líneas de código que verifican que se simula correctamente. Sim-bpred simula igual que sim-fast pero a parte de guardar datos de registros también almacena datos del predictor de saltos al igual que permite modificar su configuración antes de simular. Sim-outorder es el simulador mas completo, incluye todas las propiedades de los anteriores (excepto sim-eio) y además añade otros datos de relevancia como memoria cache entre otros.

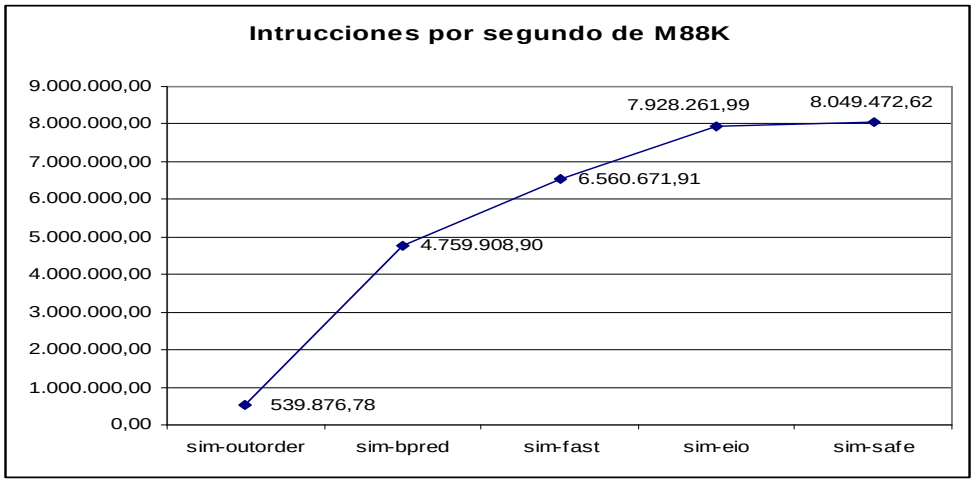
Tal i como se puede apreciar en las gráficas 4.3 conforme va decrementando la complejidad del simulador va aumentando el número de instrucciones que se pueden simular cada segundo.

A





C



4.3-A Análisis del numero de instrucciones por unidad de tiempo simuladas por la aplicación GO y con cada simulador

4.3-B Análisis del numero de instrucciones por unidad de tiempo simuladas por la aplicación LI y con cada simulador

4.3-C Análisis del numero de instrucciones por unidad de tiempo simuladas por la aplicación M88K y con cada simulador

Con estos cálculos de numero de instrucciones por segundo simuladas y sabiendo que cada aplicación de benchmark tiene miles

de millones de instrucciones, podemos llegar fácilmente a la conclusión que para una sola simulación se necesitan diversas horas. Esto implica que para cada simulación cambiando la configuración del procesador tardaría días en simularse todo y obtener unos resultados que se necesitan en el menor tiempo posible.

4.5. Planteamiento del problema

Una vez hemos podido comprobar como se deben utilizar las herramientas de que se dispone, se procede a analizar el problema e intentar escoger un camino hacia una posible solución.

En un principio, lo que se podría hacer es una simulación completa de toda la aplicación a analizar en intervalos ya definidos. El tamaño de los intervalos se debe escoger en función de la diferencia que se puede observar entre la simulación continua y la simulación paralela. Para ello se deberá simular dos veces la aplicación una por cada tipo de simulación y obtener los resultados para posteriormente analizarlos.

Una vez se ha escogido el tamaño optimo del intervalo podemos empezar a trabajar en como se puede reducir el tiempo de simulación sin variar el resultado final. La prueba pasa por ir eliminando de la simulación intervalos y comprobar la media final de los intervalos simulados. Si el resultado es inferior al porcentaje de error marcado en un principio, se puede dar por buena la simulación, sino, quiere decir que los intervalos que se han suprimido son significativos para le resultado final. Durante el análisis ya se explicara más en profundidad que método se ha escogido para suprimir los intervalos de simulación.

Con el tamaño del intervalo y con un método escogido para la realización del estudio nada mas falta escoger la configuración con la que se realizara el estudio. El realizar el estudio con todas las configuraciones mostradas en las graficas anteriores seria muchísimo

tiempo, por lo que se ha decidido escoger dos configuraciones que sean representativas. Las dos configuraciones que se consideraron como significativas fueron las bimod de tamaño 64 y la bimod de tamaño 8192. Se han escogido estas dos ya que pertenecen al mismo tipo de predictor bimod y con ello se pueden comparar las mejoras que ofrece el modificar el tamaño de la tabla, al mismo tiempo que se realiza el análisis de acortar el tiempo de simulación. El tamaño 64 se escogió dado que fue el mas pequeño posible que ofrecía un resultado coherente en numero de fallos, los tamaños inferiores a 64 mostraban unas cantidades demasiado elevadas como para poderlos tener en cuenta. El tamaño 8192 se escogió dado que es el tamaño más grande posible y que respecto a tamaños superiores ofrece menor diferencia de fallos.

4.6. Estrategia a seguir

Durante el análisis del problema se estudiara uno de los posibles métodos para la mejora en el tiempo de simulación de un benchmark. Para ello se deberá escoger una configuración fija en el predictor de saltos. El hecho de escoger una configuración fija facilitará las cosas a la hora de comparar los resultados obtenidos durante el estudio. El número de configuraciones diferentes empleadas también es importante dado que a más configuraciones más tiempo se tardara en obtener un resultado.

Antes de escoger un posible camino también se debe elegir un tamaño de intervalo óptimo. El tamaño de los intervalos nos servirá para posteriormente en el análisis de la simulación escoger los intervalos que se desean eliminar. El hecho que se defina un tamaño de intervalo óptimo quiere decir que ese tamaño en una simulación paralela no ofrece diferencias muy grandes respecto a una simulación continua. Cuando se habla de diferencias muy grandes en la

simulación paralela y serie quiere decir que sean superiores a un 5%, por poner un ejemplo.

Una vez escogido el tamaño del intervalo con el que se va a trabajar se procederá a escoger un posible método para la realización del análisis.

4.7. Problemas obtenidos durante las pruebas

En el caso de la simulación en serie, el obtener los resultados de cada intervalo requería realizar una serie de simulaciones incrementales hasta llegar al tamaño deseado. Esto quiere decir, que si se pretendía obtener los resultados en intervalos de 10 millones de instrucciones desde el principio hasta los 80 millones de instrucciones, se tenían que simular los intervalos de 0 a 10, de 0 a 20, de 0 a 30,… y así sucesivamente. Una vez obtenidos estos resultados, procesándolos en una hoja de cálculo se podía obtener fácilmente el resultado del intervalo deseado restando el del anterior.

El problema de tener que ir simulando un trozo cada vez mas grande es la redundancia de simular intervalos repetidas veces, y a parte de esto, el inconveniente del la gran cantidad de tiempo que se tarda en obtener el resultado. Por ese motivo se opto por hacer unas modificaciones en el código original del simulador, de modo que se pudieran obtener todos los resultados lanzando la simulación una única vez.

```
if (!(sim_num_insn % 10000000))
{
        sim_print_stats(stderr);          // Saca estadísticas
        bpred_after_priming(pred);        // Pone a 0 estadisticas
}
```

Figura 4.4: Fragmento de código añadido al simulador simplesim

El fragmento de código de la figura 4.4 fue añadido en el fichero sim-bpred.c. Posteriormente se tuvo que recompilar para que los cambios surgieran efecto.

Otro problema que surgió fue en la simulación en paralelo. La realización de este tipo de pruebas requiere la simulación de la aplicación por duplicado, una con sim-eio para generar los checkpoints y la otra con sim-bpred para simular el predictor a partir de los checkpoints generados. El hecho de tener que simular dos veces antes de obtener los resultados requiere un cierto tiempo adicional que se debería suprimir. Otro posible problema que puede surgir es el tamaño que ocupan los ficheros de checkpoint. En este estudio, los ficheros obtenidos mediante la simulaciones con las aplicaciones go.eio y li.eio fueron relativamente pequeños (entre 1 y 5 Mb de tamaño) en comparación con los obtenidos con la aplicación m88k.eio los cuales ocupaban alrededor de los 80 Mb.

La forma más sencilla de intentar solucionar el problema fue volver a modificar el código de la aplicación sim-bpred. En este caso la modificación realizada lo único que hace es borrar el contenido de la tabla bimod, que es la configuración que se decidió utilizar para las pruebas. El código de la figura 4.5 muestra la modificación realizada.

```
if (!(sim_num_insn % 10000000)){
        sim_print_stats(stderr);            // Saca estadísticas
        bpred_after_priming(pred);          // Pone a 0 estadísticas
// Ponemos a 0 el predictor
        if (TRUE){
                flipflop=1;
                for(cnt=0;cnt<pred->dirpred.bimod->config.bimod.size;cnt++){
                        pred->dirpred.bimod->config.bimod.table[cnt]=flipflop;
                        flipflop= 3 - flipflop;
                }
        }
}
```

Figura 4.5: Fragmento de código añadido al simulador simplesim

Como se puede apreciar, el hecho de cambiar la variable la condición TRUE a FALSE en el código y volver a compilarlo nos facilita el modo de simulación a utilizar, con TRUE se simularía la parte de los checkpoints mientras que con la opción FALSE la simulación seria continua.

Durante la obtención de los resultados y análisis de los mismos se presento otro problema. Los datos se procesan en una hoja de cálculo para posteriormente poder obtener un grafico de forma rápida y sencilla, dado que un grafico es la forma más rápida de ver los resultados obtenidos. Se comenzó trabajando bajo Linux con la distribución Debian en su versión 3.0 y como paquete de ofimática se optó por utilizar inicialmente Openoffice. La hoja de calculo que pertenece a este paquete daba el problema que cuando se le introducían una cierta cantidad de datos que hiciera que creciera mucho el tamaño la hoja de calculo tardaba mucho en abrir y cerrar y por lo tanto también tardaba mucho en generar los cálculos necesarios para obtener las graficas. La solución mas viable fue la de optar por utilizar Microsoft Windows XP con el paquete Microsoft Office XP, dado que la hoja de calculo de éste paquete, Microsoft Excel, daba la velocidad necesaria para poder trabajar en condiciones. Las hojas de Excel que se han tratado para la realización del estudio tienen un tamaño de entre 30 Mb y 100 Mb. El número de operaciones mínimo que se ha tenido que realizar en un Excel es de unos 100.000. Para realizar todas las operaciones de las hojas más grandes puedes llegar a tardar hasta 4 ó 5 horas por cada una de las hojas.

5. Presentación y análisis de resultados

En este apartado se realizará el estudio y se mostrarán los resultados obtenidos. En primer lugar realizaremos un estudio previo del comportamiento de las aplicaciones. El estudio del comportamiento básicamente es, dado un tamaño de intervalo, mostrar el número de errores que comete el predictor en cada intervalo y analizar las zonas de comportamiento. En segundo lugar nos dedicaremos a buscar un tamaño de intervalo adecuado. Para encontrar el tamaño adecuado observaremos el error que se comete con cada tamaño y los compararemos entre si. Con el tamaño del intervalo escogido procederemos a la realización del método incremental.

Primeramente realizaremos el método con las aplicaciones escogidas para el estudio, que son GO, LI y M88K. Posteriormente aplicaremos a estas tres aplicaciones una mejora que consiste en simular un pequeño intervalo inicial previo a cada intervalo para llenar la historia del predictor de saltos. Esta mejora debería mejorar el error respecto a la simulación completa.

Posteriormente aplicaremos el método a cuatro aplicaciones de SPEC00 para demostrar el buen funcionamiento del método. Las aplicaciones escogidas para la demostración del método son, GCC, MCF, PARSER y TWOLF. También aplicaremos la mejora de simular un pequeño intervalo previo a cada intervalo para llenar la historia del predictor de saltos y así mejorar el resultado final.

Para finalizar se hará una propuesta de posible mejora que consiste en realizar el método en varios procesadores de forma paralela.

5.1. Análisis del comportamiento del predictor de saltos

Una vez escogido la configuración del predictor que se va a usar durante el estudio se deberá escoger el tamaño del intervalo que se va a utilizar en las simulaciones. La elección del intervalo es importante. El tamaño del intervalo influirá en el resultado final. Si el tamaño del intervalo es demasiado pequeño no da tiempo a que el predictor aprenda y por tanto no se podrá comprobar el comportamiento del predictor. Si el tamaño es demasiado grande el predictor se pasara demasiado tiempo aprendiendo por lo que la aplicación de lo aprendido quizás no llegue a hacerse. El primer tamaño escogido ha sido el de 10 millones de instrucciones porque daba juego para poder analizar el comportamiento de la aplicación. El motivo de no haber escogido 100 millones de instrucciones es que es un tamaño muy grande, y tarda mucho en simularse completamente. También se descartó el tamaño de 100 millones de instrucciones porque no refleja de forma clara el comportamiento de la aplicación.

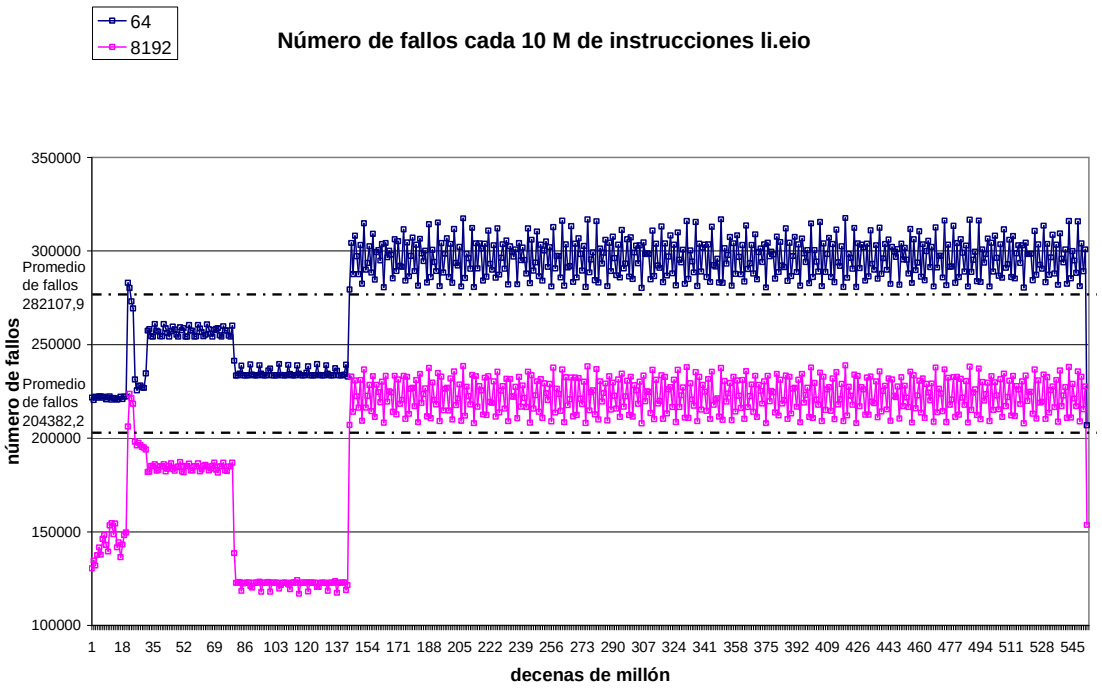


Figura 5.1: Gráfico que indica numero de fallos por intervalo de la aplicación LI.EIO simulada a intervalos de 10 millones de instrucciones.

El comportamiento de una aplicación influye a la hora de tener que escoger los intervalos a simular. La figura 5.1 muestra un gráfico del número de fallos que comete un predictor de saltos cada 10 millones de instrucciones. La aplicación empleada para esta simulación es: LI. La configuración del predictor es bimod 64 y bimod de 8192. La figura muestra el comportamiento del predictor de saltos con la aplicación LI. Se puede diferenciar cuatro fases en el comportamiento del predictor de saltos: la primera comprendida entre los intervalos 1 y 33 aproximadamente, la segunda entre los intervalos 33 y 79, la tercera entre los intervalos 80 y 143 y la cuarta entre los intervalos 144 y 553 (último). La primera zona se considera una zona inestable, puesto que dependiendo de donde se mire se pueden observar valores muy altos o valores muy bajos. La segunda zona es estable, esto quiere decir que la diferencia de amplitud entre sus valores es muy pequeña. En la tercera y cuarta zona la amplitud entre valores también es muy baja por lo que también son zonas estables. En la figura 5.1 también se puede apreciar el promedio de fallos por intervalo que se ha obtenido en la simulación. Esto servirá para determinar de donde se deben coger mas intervalos y de donde menos intervalos para obtener un resultado lo más parecido posible a la realidad. Por ejemplo, si se cogieran todos los intervalos únicamente de la zona 4, el promedio quedaría por encima del real. Por eso es importante que en esta aplicación se escoge bien el número de intervalos a simular y además que la distancia entre intervalos sea homogénea, para asegurar que cada zona recibe el peso que merece.

Veamos el comportamiento que tienen las otras dos aplicaciones al ser simuladas en las mismas condiciones. El resultado de la simulación de la aplicación GO.EIO en las mismas condiciones que la anterior se muestra en la figura 5.2. Se puede observar que el comportamiento del predictor de saltos durante la simulación de ésta aplicación es totalmente irregular durante toda su ejecución, sin

tramos definidos. No se puede dividir en zonas como pasaba con la anterior. En esta aplicación seria muy recomendable coger muchos intervalos dado que así se podría asegurar el resultado con mayor fiabilidad.

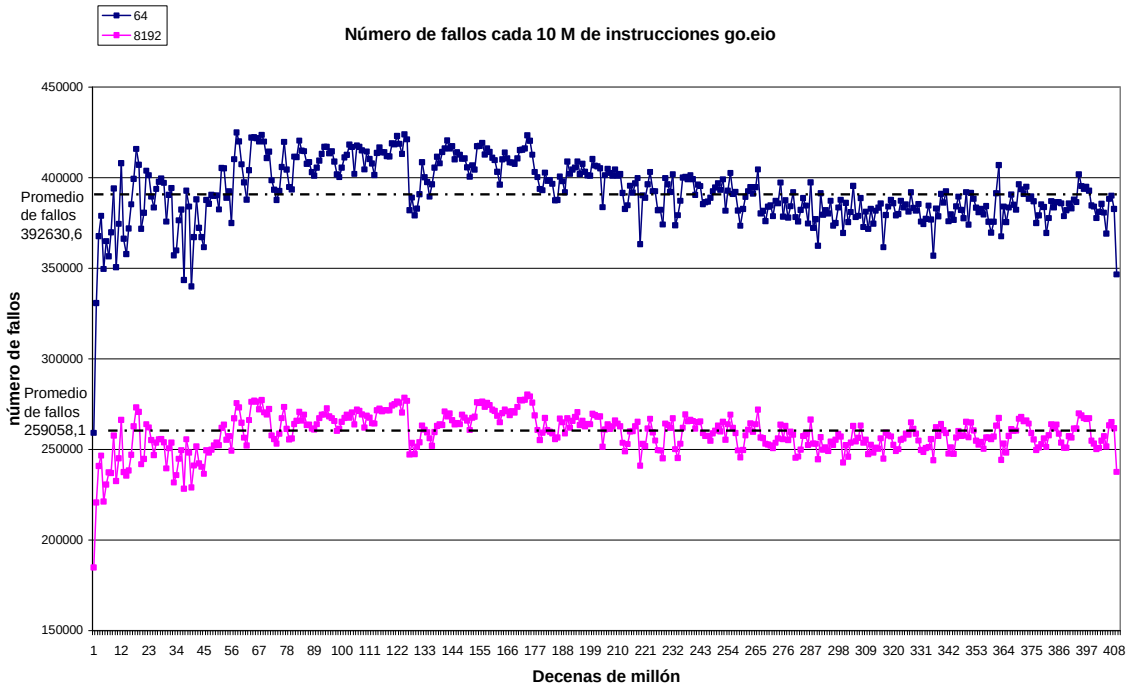


Figura 5.2: Gráfico que indica numero de fallos por intervalo de la aplicación GO.EIO simulada a intervalos de 10 millones de instrucciones.

En la gráfica de la figura 5.3 se puede comprobar que la aplicación M88K.EIO, ofrece en un principio 2 zonas diferenciadas: la primera sería la zona comprendida entre los intervalos 1 y 30 aproximadamente y la segunda zona seria la comprendida entre el intervalo 31 y el final. En la primera zona se pueden observar unas oscilaciones bastante grandes con lo que se puede decir que es una zona irregular. La segunda zona se puede considerar una zona estable, el problema que se observa es que según avanzan los intervalos el número de fallos que se comete también va aumentando de forma gradual. En esta aplicación el patrón que se debería seguir a la hora de escoger los intervalos a simular es uno del final y uno cercano al principio, con cuidado de no coger demasiados intervalos

de la zona 1, ya que es muy pequeña (solo representa el 1% de toda la aplicación) y muy irregular.

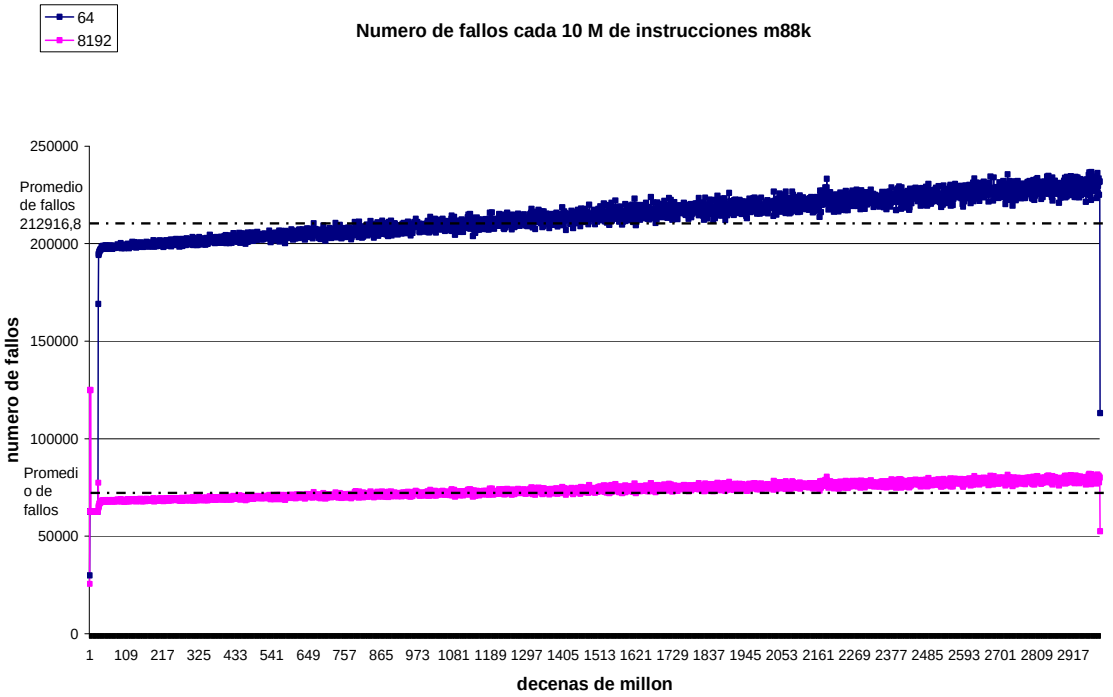


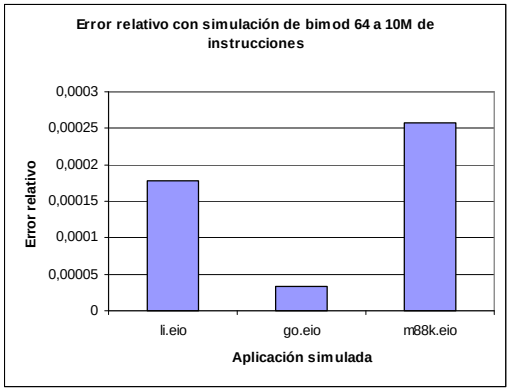
Figura 5.3: Gráfico que indica numero de fallos por intervalo de la aplicación M88K.EIO simulada a intervalos de 10 millones de instrucciones.

5.2. Elección del tamaño de los intervalos

Una vez visto el comportamiento de cada una de las aplicaciones, se ha realizado un estudio de las diferencias que se obtienen cuando se simula la aplicación de forma completa y cuando se simula la aplicación borrando el historial del predictor de saltos antes de empezar cada intervalo. Esta prueba servirá para determinar el error que se comete al empezar a simular un intervalo de cualquier parte de la aplicación sin haber pasado antes por ningún intervalo que le preceda. El resultado de esta prueba es importante ya que si escogemos un intervalo en el que la diferencia entre una la simulación completa y la simulación a trozos sea muy grande el error también lo será y por lo tanto deberemos coger muchos mas intervalos para compensarlo.

En la figura 5.4-A se muestra el error relativo calculado de la simulación de las aplicaciones GO.EIO, LI.EIO y M88K.EIO a intervalos de 10 millones de instrucciones y con la configuración del predictor bimod 64 y en la figura 5.4-B se muestra el resultado obtenido de la configuración del predictor bimod 8192. El error relativo obtenido es muy bajo.

A



B

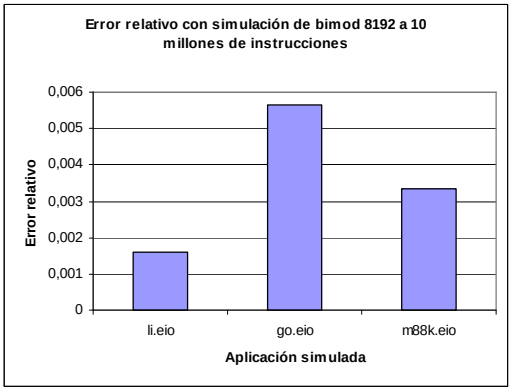


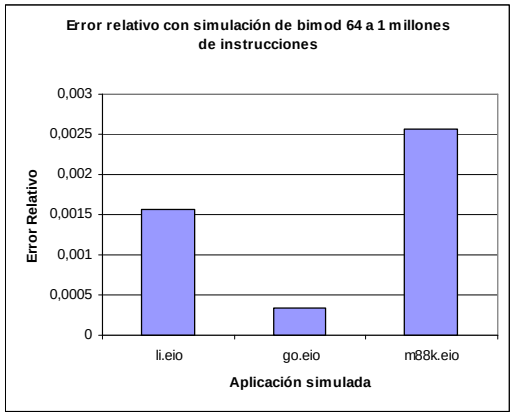
Figura 5.4-A: Gráfico que muestra el error relativo de la simulación a 10 millones de instrucciones con configuración del predictor Bimod 64. Borrando el historial en cada intervalo respecto a no borrar el historial.
B: Gráfico que muestra el error relativo de la simulación a 10 millones de instrucciones con configuración del predictor Bimod 8192. Borrando el historial en cada intervalo respecto a no borrar el historial.

Con estos datos solo podemos saber que la diferencia entre la simulación completa y la simulación a trozos a intervalos de 10 millones de instrucciones es muy pequeña. Para poder decidir si este es el mejor tamaño de intervalo o no, se necesita comparar con más simulaciones a distintos intervalos. Veamos que pasa si reducimos el tamaño del intervalo.

El comportamiento de las tres aplicaciones es el mismo en cada una de las simulaciones realizadas a distintos tamaños. Tal y como muestra la gráfica de la figura 5.5-A, el error relativo se aprecia en la diferencia de fallos por intervalo en las simulaciones a intervalos de 1 millón de instrucciones es claramente superior. El error relativo a intervalos de 1 millón es 10 veces mayor que el obtenido en intervalos de 10 millones. En la figura 5.5-B se muestra el mismo grafico obtenido de la simulación de las aplicaciones con la segunda configuración del predictor de saltos. En la simulación con

configuración del predictor bimod 8192 el aumento en el error relativo no ha sido tan pronunciado como el anterior, pero igualmente ha sido bastante notable, entorno a 8 veces el error anterior.

A



B

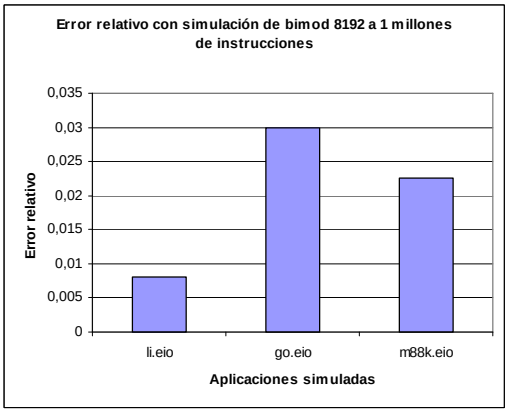
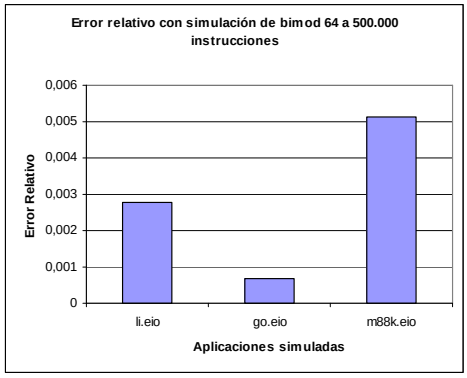


Figura 5.5-A: Gráfico que muestra el error relativo de la simulación a 1 millones de instrucciones con configuración del predictor Bimod 64. Borrando el historial en cada intervalo respecto a no borrar el historial.
B: Gráfico que muestra el error relativo de la simulación a 1 millones de instrucciones con configuración del predictor Bimod 8192. Borrando el historial en cada intervalo respecto a no borrar el historial.

El error relativo ha sido incrementado sustancialmente al reducir el tamaño del intervalo. Este incremento es debido a que el número de intervalos ha sido incrementado en 10 veces. Por ejemplo, la aplicación li.eio ha pasado de simular 553 intervalos a simular 5528 intervalos. El simular más intervalos quiere decir que el simulador se ha incrementado el número de veces que se ha borrado el historial creado y, por lo tanto, que se ha tenido que rehacer.

Vamos a continuar examinando tamaños mas pequeños para comprobar si podemos reducir aun mas el tamaño de los intervalos. El hacer el tamaño más pequeño permitirá coger más muestras de momentos distintos de la aplicación, por lo que permitirá sacar un resultado más ajustado al resultado que se obtendría si se simulara por completo. La siguiente prueba se realiza con simulaciones a intervalos de 500.000 instrucciones. Tal y como se puede apreciar en las dos graficas de la figura 5.6 el error relativo se ha prácticamente doblado en todos los casos respecto al tamaño anterior (1 millón de instrucciones).

A



B

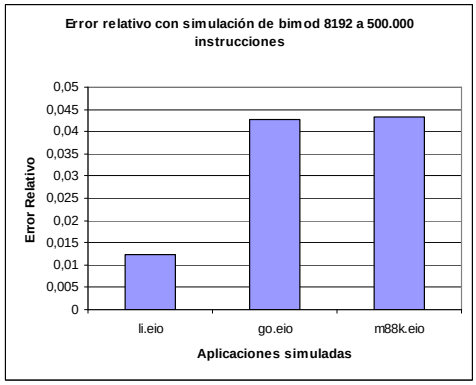
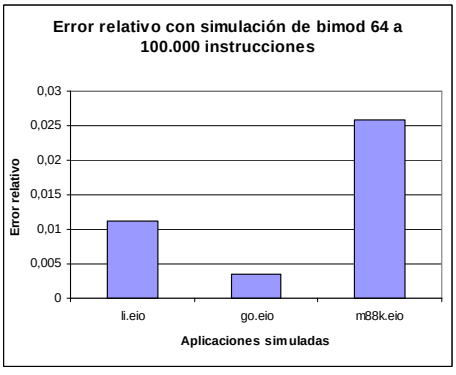


Figura 5.6-A: Gráfico que muestra el error relativo de la simulación a 100.000 instrucciones con configuración del predictor Bimod 64. Borrando el historial en cada intervalo respecto a no borrar el historial.
B: Gráfico que muestra el error relativo de la simulación a 100.000 instrucciones con configuración del predictor Bimod 8192. Borrando el historial en cada intervalo respecto a no borrar el historial.

Las siguientes graficas que se mostraran serán de la simulación de las aplicaciones li, go y m88k a intervalos de 100.000 instrucciones. La tendencia es que en las siguientes gráficas se muestre un incremento en el error relativo. En la figura 5.7-A se muestra el gráfico del error relativo de la simulación con la configuración del predictor de saltos bimod 64 y cogiendo intervalos de 100.000 instrucciones. Los incrementos que se aprecian respecto a la muestra de 500.000 instrucciones son los esperados. Se prácticamente dobla el error relativo. La grafica 5.7-B en cambio, la aplicación m88k incrementa su error relativo de forma espectacular. El error relativo en intervalos de 100.000 instrucciones se multiplica por prácticamente 5. El valor que recoge este error relativo se hace tan grande que no serviría para realizar el estudio.

A



B

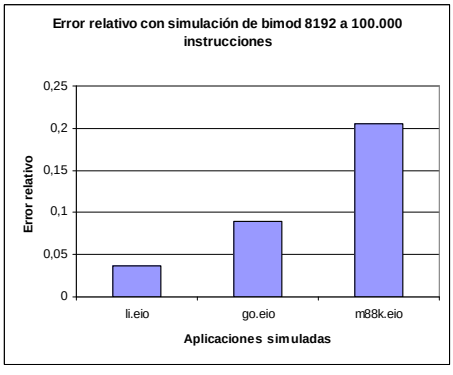


Figura 5.7-A: Gráfico que muestra el error relativo de la simulación de 100.000 instrucciones con configuración del predictor Bimod 64. Borrando el historial respecto a no borrar el historial.
B: Gráfico que muestra el error relativo de la simulación a 100.000 instrucciones con configuración del predictor Bimod 8192. Borrando el historial respecto a no borrar el historial.

Una vez vistos los errores relativos a distintos tamaños es el momento de escoger un tamaño que mantendremos constante durante todo el estudio y que permitirá sacar unos resultados fiables siempre dentro del error relativo que nos marcamos.

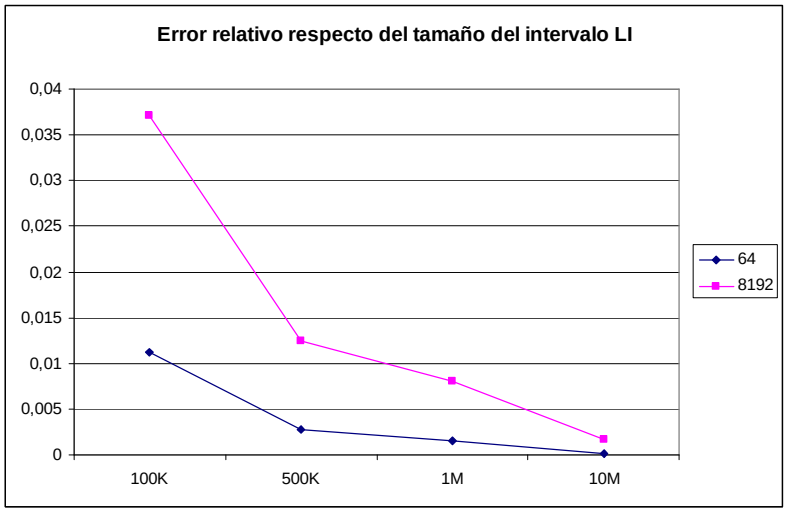
Con los resultados obtenidos hasta ahora ya tenemos información suficiente para poder escoger un tamaño de intervalo adecuado. El tamaño de intervalo de 10 millones de instrucciones es el que ofrece menos error relativo, también era de esperar, pero resulta ser demasiado grande. Recordemos también que buscamos poder coger muestras de todas las partes de la simulación de forma equitativa para poder obtener un resultado lo mas cercano posible a la simulación completa. Por lo tanto el intervalo de 10 millones de instrucciones lo descartamos por ser tan grande que no podemos seleccionar intervalos de las diferentes partes de la simulación de forma equitativa.

El tamaño intervalo de 1 millón de instrucciones también tiene un error relativo muy bajo aunque ligeramente superior al tamaño de 10 millones de instrucciones. La ventaja de este tamaño es que si permite hacer juego con el numero de intervalos a coger. Por ese motivo este tamaño de intervalo puede ser el que nos interese.

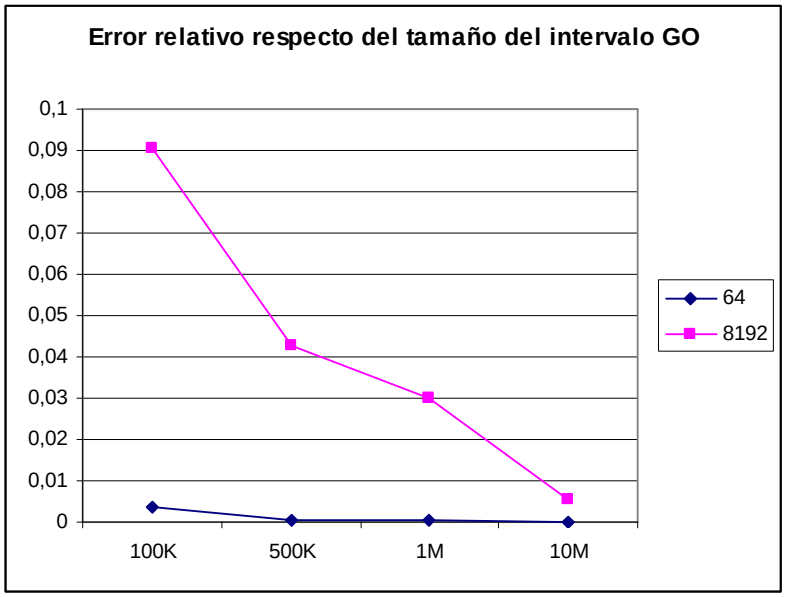
En el análisis de los intervalos de tamaño 500.000 instrucciones se aprecia que el error relativo es el esperado dada la disminución de instrucciones respecto del intervalo de 1 millón y 10 millones de instrucciones. Esta opción también podría ser valida si decidiéramos escoger este tamaño de intervalo. Es un tamaño de intervalo que ofrece un error relativo más bien bajo y el número de intervalos que podemos escoger es elevado.

Por ultimo el tamaño de 100.000 de instrucciones no es óptimo. El primer inconveniente que se encuentra es el increíble aumento en el error relativo. Esto se debe a que el historial del predictor de saltos no tiene tiempo suficiente para aprender de los saltos anteriores antes de empezar de cero nuevamente.

A



B



C

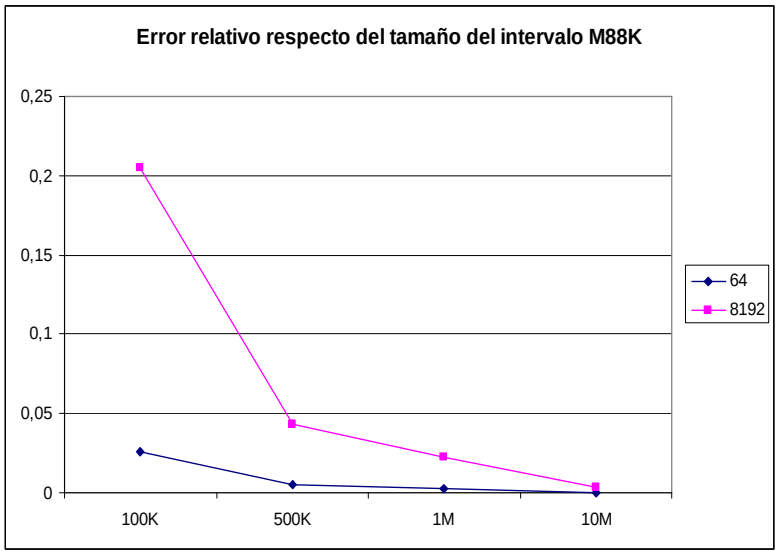


Figura 5.8-A: Tendencia del error relativo respecto al tamaño del intervalo en la aplicación LI
B: Tendencia del error relativo respecto al tamaño del intervalo en la aplicación GO
C: Tendencia del error relativo respecto al tamaño del intervalo en la aplicación M88K

Tal y como se muestra en la figura 5.8, el tamaño de 100.000 instrucciones no es adecuado para realizar el estudio dado que partiríamos de un error relativo muy grande con lo que tendríamos poco margen para pode errar más. Por el contrario, entre los tamaños de 1 millón de instrucciones y 500.000 instrucciones, escogeremos finalmente el de 1 millón de instrucciones ya que por error relativo es el que mejor resultado nos ofrece.

5.3. Análisis Detallado del Método de Simulación Incremental

En este apartado se expondrán los resultados obtenidos durante el estudio. Y se harán las comparaciones pertinentes para comprobar si los resultados obtenidos se ajustan a lo esperado en un principio.

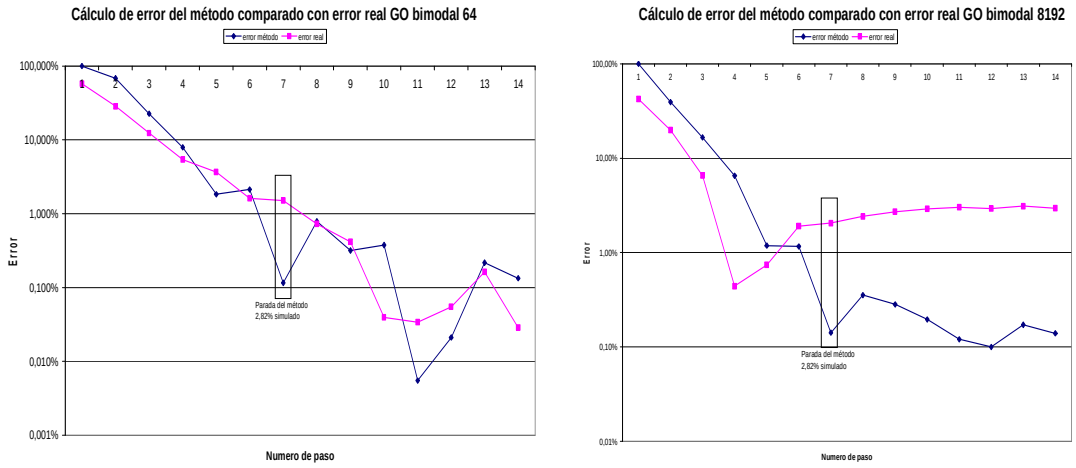
En este apartado se empezaran a ver los resultados obtenidos durante las pruebas realizadas con los programas M88K, LI y GO. El método incremental fue el que se utilizó para la obtención de los resultados. El método incremental (3.1) consiste en, básicamente, dividir la aplicación en intervalos y pasos. Cada paso significa simular varios intervalos de la aplicación. Los intervalos escogidos empezaban por el centro y en cada paso se dividía cada parte de la aplicación por simular y se cogen los intervalos del centro. De este modo se van repitiendo pasos hasta que el error respecto al paso anterior es inferior a un cierto valor, error, escogido antes de empezar. El tamaño del intervalo (5.3) se ha determinado en 1 millón de instrucciones. También recordaremos que la configuración del simulador (2.7) es un tipo "bimodal" de 64 y 8192 bits. Se escogieron estos valores dado que inferior a 64 empezaba a fallar mucho el predictor de saltos y mayor de 8192 no se apreciaba la diferencia con un valor mayor. Ahora es cuando llega la hora de escoger el valor que nos servirá de referencia para detener los pasos. Tengamos en cuenta que el hecho de haber dividido en intervalos la simulación

hace que haya un error irreparable. El error aparece cada vez que empezamos a simular un intervalo. Al empezar a simular un intervalo lo hacemos con el histórico del predictor de saltos vació, mientras que si el intervalo se cogiera teniendo en cuenta el histórico de los intervalos que lo preceden el error seria menor que el que nos acabara dando. Por lo que el valor ε que escogemos para la realización del estudio será 1%. Recordaremos que el valor de ε en cada paso se calcula:

$$Error\ Actual = \left(\frac{Error\ en\ el\ paso\ anterior}{Error\ en\ el\ paso\ actual} - 1 \right) \times 100$$

Figura 5.9: Formula que calcula el error relativo entre el paso actual y el anterior en la simulación incremental.

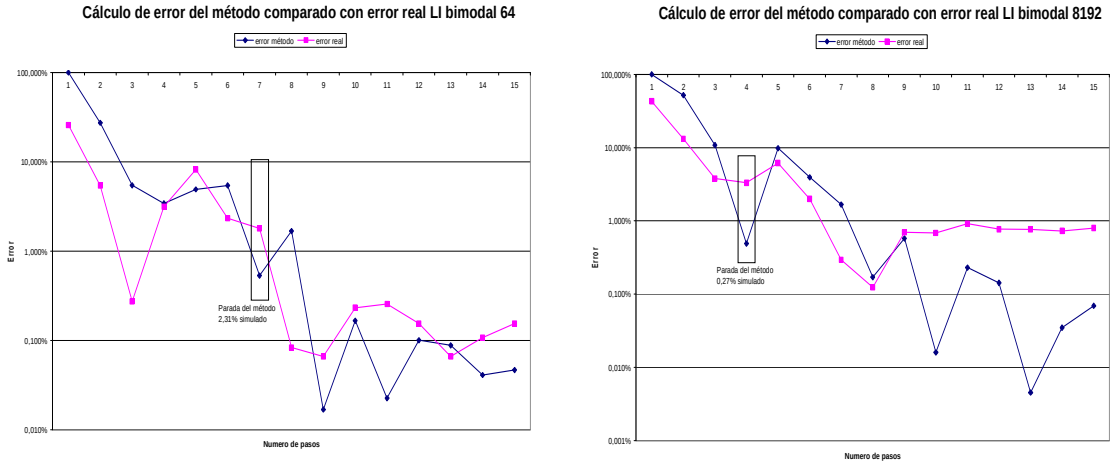
Una vez recordados los datos del estudio empezamos a ver los resultados que se obtuvieron al realizar las pruebas. Empezamos viendo los resultados de la aplicación GO. Recordaremos que GO es la aplicación más corta de las tres que se han utilizado para el estudio. En nuestro estudio hay una pequeña diferencia respecto la idea original del método. Nuestro método comienza por el primer intervalo, el cual siempre se simula. En el segundo paso se simula el intervalo a distancia 2^n y en el tercer paso se simulan los intervalos a distancias de 2^{n-1}, y así sucesivamente. Se empieza por el primer intervalo dado que consideramos que era un intervalo importante a ser simulado. Se considera importante la simulación del primer intervalo porque es una parte de la aplicación que raramente tiene un comportamiento uniforme con respecto al resto. Esto es debido a que en la parte inicial hay definición de variables y funciones y sus inicializaciones. El resultado obtenido y que después se ira analizando es el numero de fallos del predictor de saltos obtenidos en cada intervalo. El resultado de cada paso es el promedio que se obtiene del número de fallos de cada intervalo.



5.10-A Gráfico que muestra la progresión en cada paso del error relativo respecto del paso anterior y el error real que se comete con la aplicación GO y configuración bimodal de 64.
B Gráfico que muestra la progresión en cada paso del error relativo respecto del paso anterior y el error real que se comete con la aplicación GO y configuración bimodal de 8192.

En la grafica 5.10-A se muestra la progresión por pasos del método comparado con la progresión real que se obtendría del error. La aplicación que se simula en la figura 5.10 es GO con configuración bimodal de 64 bits en A y bimodal de 8192 bits en B. En el primer paso del método el valor es uno puesto que no hay un paso anterior respecto al que compararlo. A partir del paso dos es cuando se comienza a comparar con el paso anterior y es cuando se empieza a obtener un resultado. La grafica, como se preveía es decreciente. Se puede observar que en la figura 5.10-A el comportamiento del método se asemeja bastante al error real cometido. En la gráfica 5.10-B el comportamiento del método respecto al error real empieza a distanciarse en el paso 6, donde el método disminuye el error en cada paso mientras el error real va incrementando.

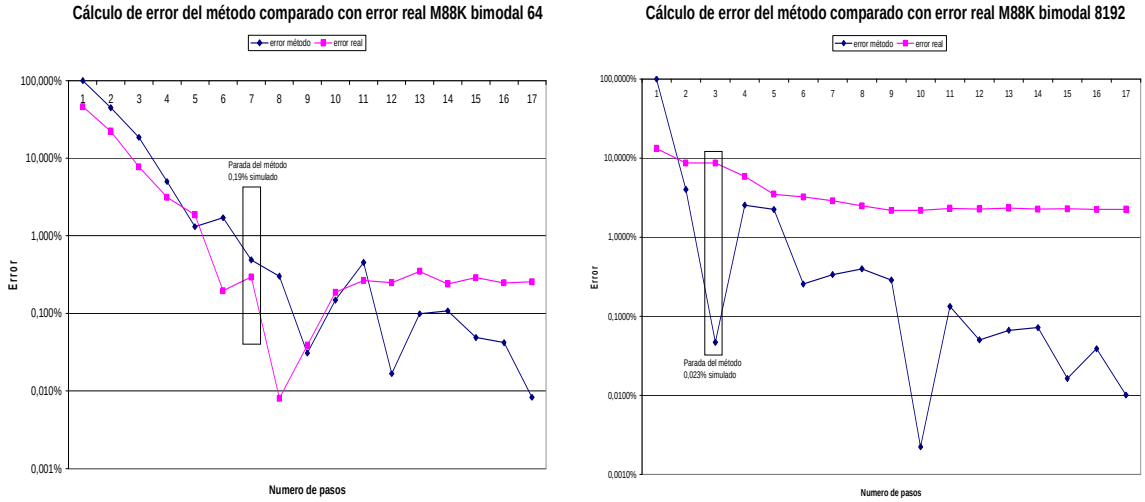
En el paso 7 es donde nuestro método se detiene, tal y como se muestra en la figura 5.10-A. En la figura 5.10-B el método se detiene en el mismo punto. Si calculamos el número de instrucciones simulado hasta el punto de parada del método, observamos que se han simulado 127 intervalos. Si sabemos que la aplicación GO es de 4.500 millones de instrucciones entonces se puede decir que obtenemos un resultado simulando tan solo un 2,82% de la aplicación.



5.11-A Gráfico que muestra la progresión en cada paso del error relativo respecto del paso anterior y el error real que se comete con la aplicación LI y configuración bimodal de 64.
B Gráfico que muestra la progresión en cada paso del error relativo respecto del paso anterior y el error real que se comete con la aplicación LI y configuración bimodal de 8192.

La aplicación que se simula en la figura 5.11 es LI con configuración bimodal de 64 bits en A y bimodal de 8192 bits en B. En la grafica 5.11-A se puede observar que mantiene una progresión descendiente, igual que pasaba en las gráficas anteriores. El método detiene la simulación en el paso 7 en la simulación bimodal de 64, gráfica 5.11-A. En la figura 5.11-A el error real y el error en cada paso del método es bastante similar. La gráfica 5.11-B empieza con un comportamiento del error real parejo al del error en cada paso del método. Es a partir del paso 9 donde el método y el error real se van distanciando. Mientras el error dado por el método tiene altibajos, el error real de la simulación se mantiene estable.

En la grafica 5.11-B el comportamiento inicial es más acentuado, esto provoca que el método se pare en el paso 4 pese a que si hubiéramos seguido simulando, el error vuelve a llegar al 10% respecto al paso anterior. Parando en el paso 7 y sabiendo que la aplicación LI tiene 5.500 millones de instrucciones, podemos calcular que simulamos un 2,31% de la aplicación mientras que parando en el paso 4 obtenemos un resultado con tan solo un 0,27% de la aplicación simulada.



5.12-A Gráfico que muestra la progresión en cada paso del error relativo respecto del paso anterior y el error real que se comete con la aplicación M88K y configuración bimodal de 64.
B Gráfico que muestra la progresión en cada paso del error relativo respecto del paso anterior y el error real que se comete con la aplicación M88K y configuración bimodal de 8192.

La aplicación que se simula en la figura 5.12 es M88K con configuración bimodal de 64 bits en A y bimodal de 8192 bits en B. Las graficas 5.12-A y 5.12-B, igual que pasaba con las anteriores, tienen un comportamiento decreciente. También se puede observar un ligero aumento del valor de error en la grafica 5.12-B. La grafica 5.12-A muestra un error del método muy parecido al error real. En la figura 5.12-B el error real no tiene ningún parecido al error dado por el método. Mientras el error real se mantiene estable en una línea casi horizontal el método va teniendo altibajos hasta el final.

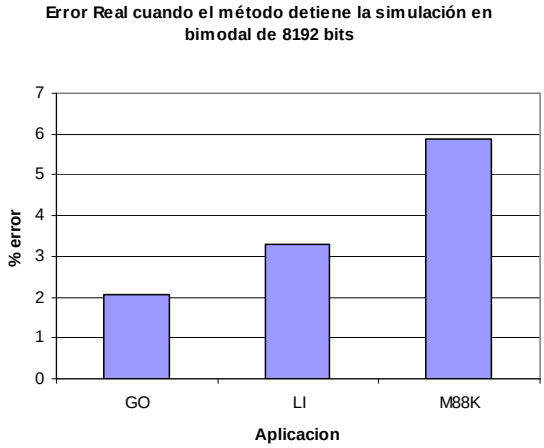
El método se detiene en el paso 7 mientras que la grafica de la figura 5.12-B se detiene en el paso 3. El número de intervalos simulados en el paso siete en esta simulación son 58 y por tanto el porcentaje simulado respecto a toda la simulación es de un 0,19%. El porcentaje de simulación deteniendo el proceso en el paso 3 es del 0,023%.

Con estos datos podemos observar el enorme ahorro de simulación que se puede tener para obtener un resultado. Ahora vamos a comprobar si realmente ese resultado que se obtiene es adecuado para trabajar. Para comprobar si el resultado obtenido es adecuado se han simulado las aplicaciones completamente y se han

obtenido unos resultados. Las simulaciones que se han hecho han sido de dos tipos. La primera ha sido una simulación a intervalos pero sin eliminar el histórico del predictor de saltos. De esta forma obtenemos el resultado con menor error posible. Posteriormente se simula toda la aplicación borrando el histórico al comienzo de cada intervalo. De esta forma sabemos a partir de que error nos movemos. Luego analizamos gráficamente el resultado para ver si detenemos correctamente la simulación.

A

B



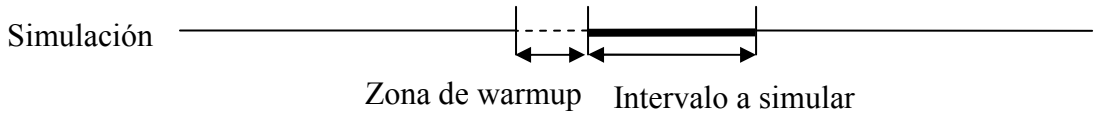
5.13-A Gráfica que muestra el error existente en guardar o no guardar el histórico en la simulación a intervalos. Configuración del predictor bimodal de 64 bits.
B: Gráfica que muestra el error existente en guardar o no guardar el histórico en la simulación a intervalos. Configuración del predictor bimodal de 8192 bits.

Como se puede observar en las gráficas 5.13-A el error existente en la simulación de M88K es bastante pequeño pese a haber simulado muy pocos intervalos respecto a la simulación completa, mientras que en GO y LI el error es bastante mayor. Esto se debe a que en M88K se ha llegado a coger los intervalos significativos que darían con el comportamiento de la aplicación. LI y GO no sólo han tardado más en encontrar los intervalos significativos sino que el error cometido es mayor. En cambio en la figura 5.13-B el error cometido es mucho mayor en M88k. El que la configuración bimodal de 8192 bits de un error mucho mayor que el de bimodal de 64 bits es debido a que al predictor le cuesta mas aprender con una historia más grande por lo que comete más errores.

Ahora procederemos a comprobar el error que se comete en cada uno de los pasos de la simulación completa borrando histórico respecto a la simulación completa sin borrar histórico.

5.4. Warmup

En este apartado se tratará el hecho de crear una pequeña historia que preceda la simulación del intervalo. Esta pequeña historia perjudica en el tiempo que se tarda en simular el intervalo, pues también se necesita tiempo para simular la historia. Se ha considerado que el fragmento de historia que se debe simular sea de alrededor del 10% del tamaño del intervalo.



5.14: Ejemplo de la zona de warmup

En la figura 5.14 se puede observar gráficamente que es la zona de warmup exagerando las proporciones para que se vea de forma mas clara. La zona punteada de la figura 5.14 es la zona de calentamiento donde únicamente se obtiene información para rellenar el histórico del predictor de saltos. Justo cuando termina el calentamiento se tuvo que poner a cero las estadísticas del preditor de saltos. Con las estadísticas a de fallos en el predictor a cero y le histórico con información empieza el intervalo (zona gruesa de la figura 5.14).

El tamaño de este pequeño intervalo es complicado de escoger con exactitud. Si coges un tamaño muy grande, llega un punto en que la historia del predictor no prende más y si te quedas corto habrá tan poca historia que la mejora casi ni se notará. Únicamente se han realizado pruebas con el tamaño de 100.000 instrucciones (el 10% de 1 millón de instrucciones). Pese a no haber realizado más pruebas con otros intervalos mayores o menores no podemos comparar si se

ha escogido bien o mal. Únicamente se mostraran los resultados comparados con los obtenidos en el apartado 5.5 respecto a una simulación sin perdida de historia para verificar que realmente funciona y se obtiene una mejora.

A B

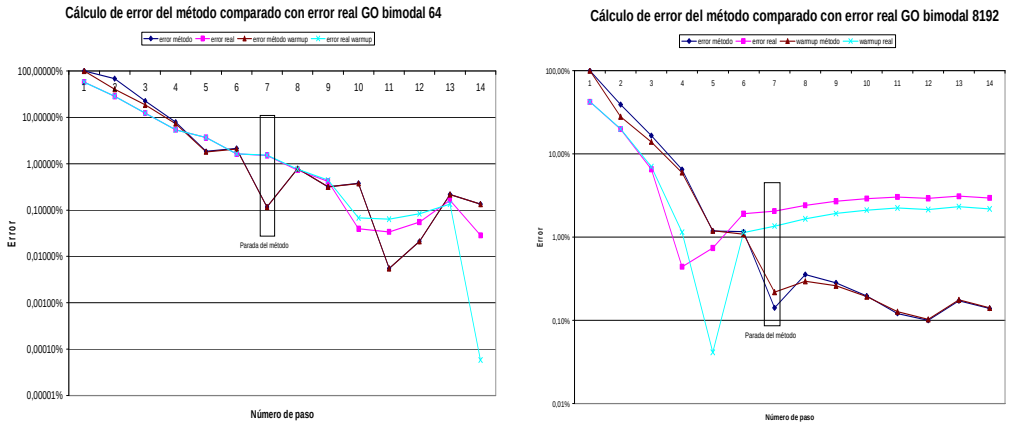
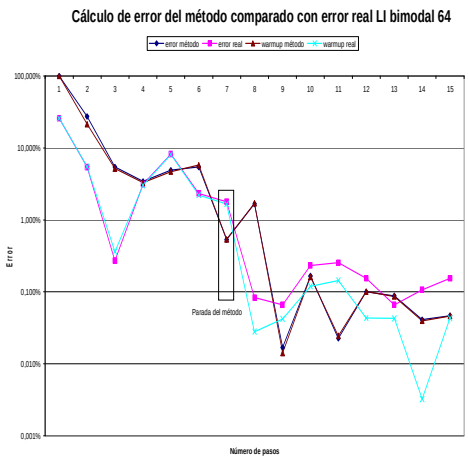


Figura 5.15-A: Gráfica que muestra la diferencia del método incremental paso a paso simulado con warmup y sin warmup y los errores reales de cada uno para GO Bimodal de 64.
B: Gráfica que muestra la diferencia del método incremental paso a paso simulado con warmup y sin warmup y los errores reales de cada uno para GO Bimodal de 8192.

La figura 5.15 muestra la diferencia en el método y el error real cometido por la aplicación GO aplicándole un calentamiento inicial. Podemos observar que el método se detiene exactamente en el mismo paso que en el caso anterior, el paso 7. Tanto en A como en B se puede observar que el error dado por el método no tiene prácticamente diferencia, por lo que el calentamiento inicial parece que surta poco efecto. En cambio en el error real si se puede observar una ligera diferencia. La diferencia se hace mas notable en la figura 5.15-B. Esto es debido a que la historia del predictor de saltos de la figura 5.15-B es mucho más grande que el de la figura A y por tanto con un pequeño calentamiento inicial puede mejorar sustancialmente el resultado.

A



B

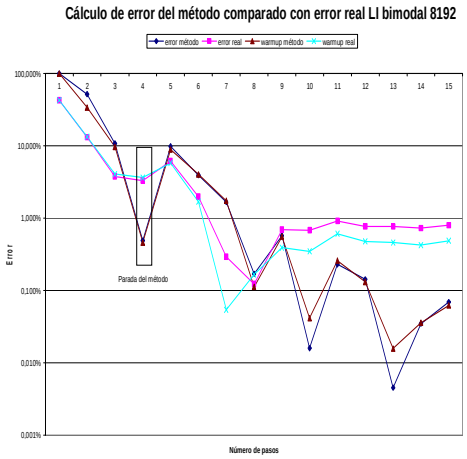
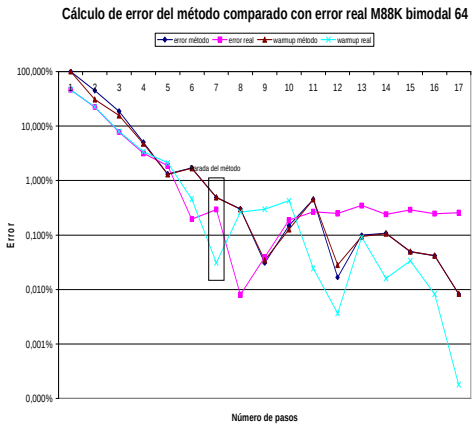


Figura 5.16-A: Gráfica que muestra la diferencia del método incremental paso a paso simulado con warmup y sin warmup y los errores reales de cada uno para LI Bimodal de 64.
B: Gráfica que muestra la diferencia del método incremental paso a paso simulado con warmup y sin warmup y los errores reales de cada uno para LI Bimodal de 8192.

La figura 5.16 muestra la diferencia en el método y el error real cometido por la aplicación LI aplicándole un calentamiento inicial. Se puede observar que el método se detiene en el mismo punto tanto para la configuración bimodal de 64 bits (figura 5.16-A) como para bimodal de 8192 bits (figura 5.16-B). Al igual que pasaba en la figura anterior la diferencia en el método con el calentamiento y sin el es mínima. En cambio la diferencia del error real cometido se ve bastante mejorada en ambas configuraciones.

A



B

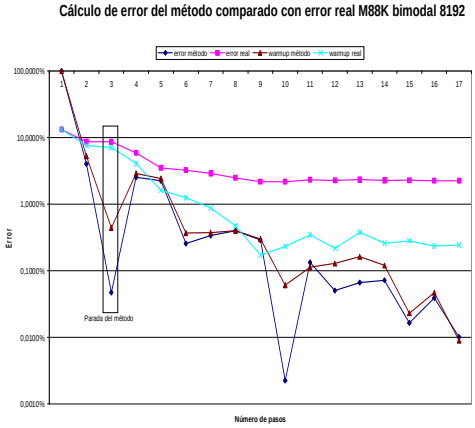
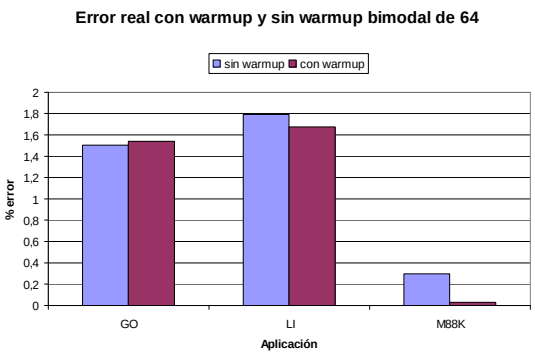


Figura 5.17-A: Gráfica que muestra la diferencia del método incremental paso a paso simulado con warmup y sin warmup y los errores reales de cada uno para M88K Bimodal de 64.
B: Gráfica que muestra la diferencia del método incremental paso a paso simulado con warmup y sin warmup y los errores reales de cada uno para M88K Bimodal de 8192.

La figura 5.17 muestra la diferencia en el método y el error real cometido por la aplicación M88K aplicándole un calentamiento inicial. Igual que pasaba en las anteriores figuras en la figura 5.17-A el método parece no tener una mejora por aplicarle un pequeño calentamiento. En la figura 5.17-B el método si que presenta una mejora visible. Se puede observar que el error del método con calentamiento es siempre inferior respecto a la del método sin calentamiento exceptuando momentos puntuales. El error real también se ve reducido en ambas gráficas. Como dato curioso se puede comentar que en los pasos 7, 8 y 9 de la figura 5.17-A el método con calentamiento obtiene un índice de error real bastante mayor al de la simulación sin calentamiento.

A



B

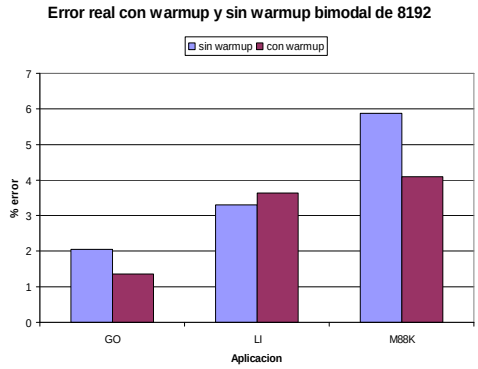


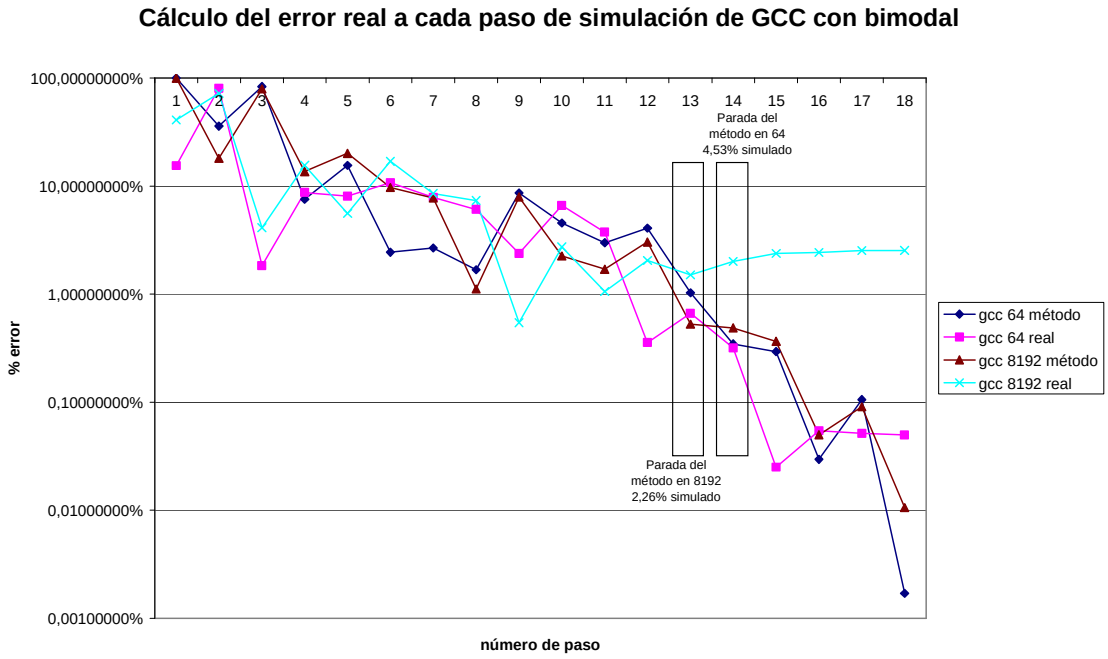
Figura 5.18-A: Gráfica que muestra el error que se obtiene al detener la simulación en el punto que marca el método para cada aplicación con configuración bimodal de 64 con y sin warmup.
B: Gráfica que muestra el error real que se obtiene al detener la simulación en el punto que marca el método para cada aplicación con configuración bimodal de 8192 con y sin warmup.

Las graficas de la figura 5.18 confirman que con un pequeño warmup se mejora el resultado en el error final. En las aplicaciones GO y LI la diferencia es mucho menor que la que muestra la aplicación M88K. Esto se debe a que M88K es prácticamente 5 veces más larga que GO y LI. Cuanto más larga sea la aplicación que se simula más se ve la diferencia dado que se simulan más intervalos.

5.5. Comprobación del Método de Simulación Incremental

En este apartado se intentará demostrar con una serie de aplicaciones de SPEC00 que el método es generalizable. Para intentar demostrarlo se analizaran gráficas similares a las expuestas en el apartado anterior. Posteriormente se realizará el estudio de aplicar un calentamiento inicial al método y se analizarán los resultados.

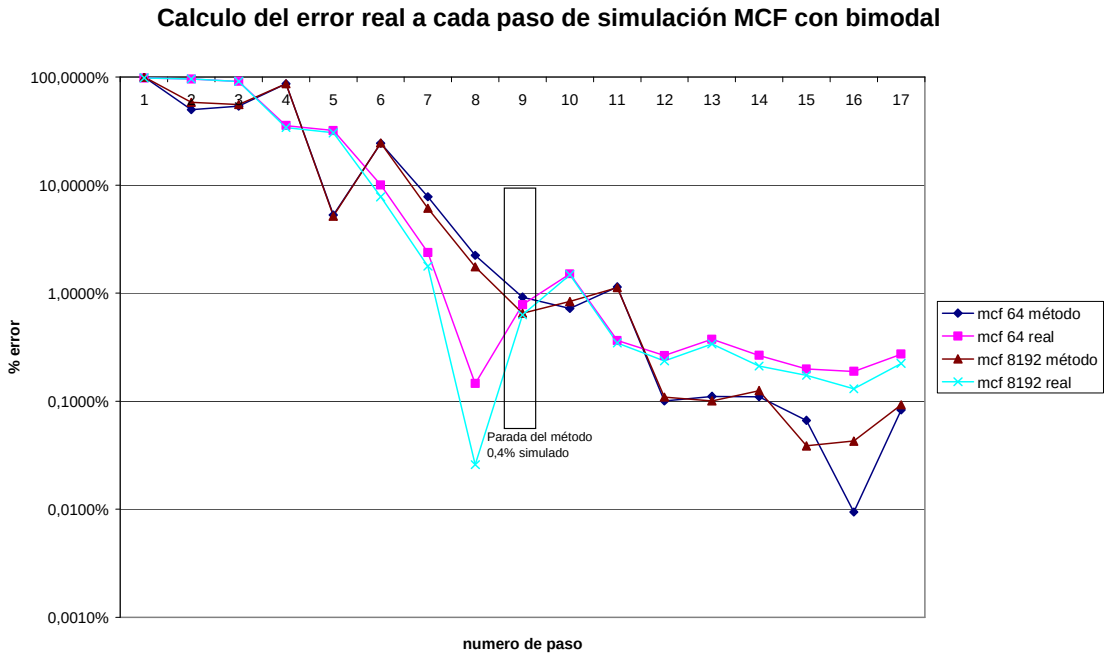
5.5.1. Sin warmup



5.19 Gráfico que muestra la progresión en cada paso del error relativo respecto del paso anterior. Configuración bimodal de 64 y 8192. Aplicación GCC.

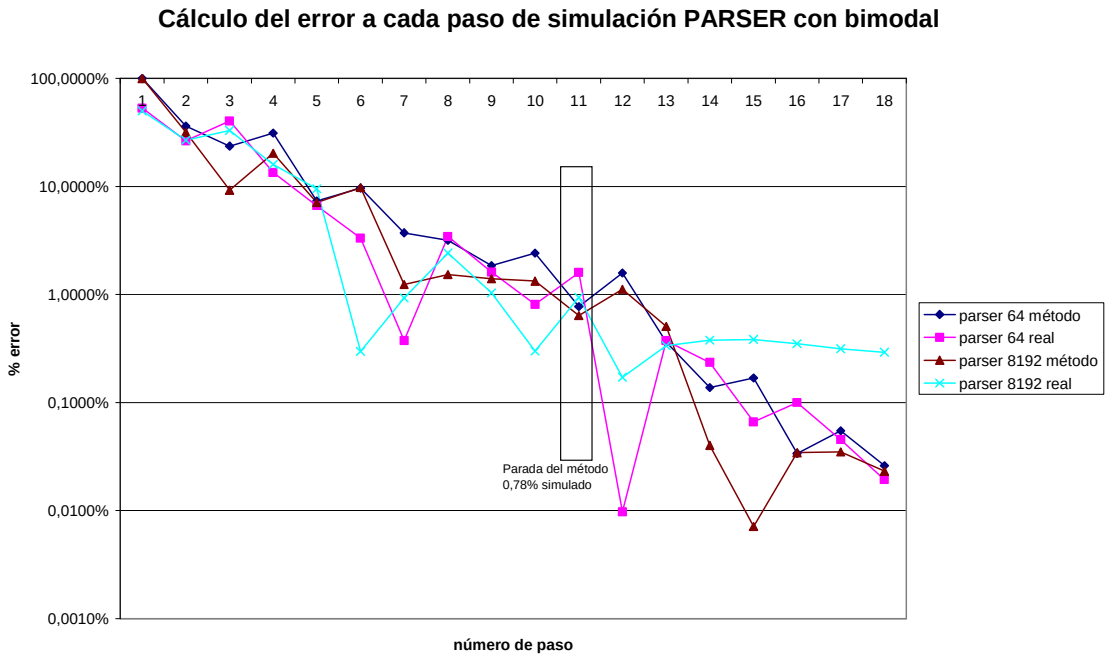
La gráfica 5.19 muestra el comportamiento de la aplicación GCC con las dos configuraciones, bimodal de 64 bits y bimodal de 8192 bits. Según muestra la gráfica el método se detiene en los pasos 14 y 13 para bimodal de 64 bits y bimodal de 8192 respectivamente. En el paso 13 se han simulado 2459 intervalos, si la aplicación tiene 108613 intervalos, esto implica que hemos simulado un 2,26% del computo global. Mientras que en el paso 14 se han simulado 4916, que quiere decir que se ha simulado el 4,53%. Como se puede observar, el número de instrucciones simuladas sigue siendo bastante bajo. Por lo que se puede observar en la gráfica de la figura 5.19 el

error real cometido en el punto de parada del método también es bastante bajo.



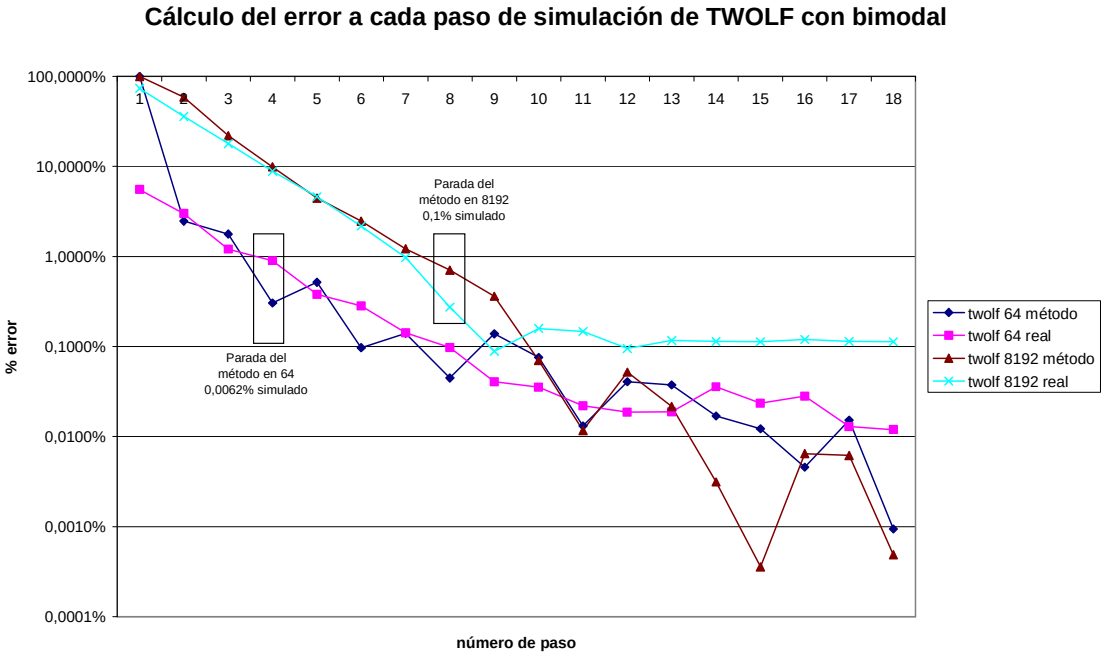
5.20 Gráfico que muestra la progresión en cada paso del error relativo respecto del paso anterior. Configuración bimodal de 64 y 8192. Aplicación MCF.

La gráfica 5.19 muestra el comportamiento de la aplicación MFC con las dos configuraciones, bimodal de 64 bits y bimodal de 8192 bits. Como podemos observar en la figura 5.20 el método incremental detiene la simulación en el paso 9 en ambas configuraciones. Como curiosidad también se puede destacar que el error del método en ese punto es muy parecido en ambas configuraciones y también es muy parecido al error real que da la aplicación. La detención en el paso 9 quiere decir que se han simulado hasta el momento 250 intervalos. La aplicación MFC consta de 61868 intervalos de un millón de instrucciones, por lo que se ha simulado un 0,4% de la aplicación. Con tan solo un 0,4% de la aplicación simulado se obtiene un error real de menos de 1%, según muestra la gráfica 5.20.



5.21 Gráfico que muestra la progresión en cada paso del error relativo respecto del paso anterior. Configuración bimodal de 64 y 8192. Aplicación PARSER.

La gráfica 5.19 muestra el comportamiento de la aplicación PARSER con las dos configuraciones, bimodal de 64 bits y bimodal de 8192 bits. Igual que pasaba con la aplicación MCF expuesta en la figura 5.20, la aplicación PARSER mostrada en la figura 5.21 muestra unas líneas de error muy parejas. Tanto el error cometido en el método incremental como el error que se comete realmente están muy próximos. No es de extrañar pues que para ambas configuraciones del predictor de saltos el método detenga la simulación en el mismo punto, el paso 11. Parser es una aplicación de 114478 intervalos de los que tan solo se han simulado 895, por lo que se ha simulado un 0,78% para obtener el resultado.



5.22 Gráfico que muestra la progresión en cada paso del error relativo respecto del paso anterior. Configuración bimodal de 64 y 8192. Aplicación TWOLF.

La gráfica 5.19 muestra el comportamiento de la aplicación GCC con las dos configuraciones, bimodal de 64 bits y bimodal de 8192 bits. La gráfica del método mantiene un comportamiento distinto según el tamaño de la historia. Una historia más pequeña inicialmente parece dar menor error que una mayor, pero conforme se van simulando intervalos, parece que se invierte la situación. La configuración bimodal de 64 bits al cometer un error menor inicialmente, se detiene antes, en el paso 4. La configuración de 8192 bits en cambio, se detiene en el paso 8. Esto quiere decir, que para la primera configuración hemos simulado 7 intervalos de los 113726 que tiene la aplicación Twolf, que supone un 0,0062%. Y para la configuración mayor, simulamos 112 intervalos, que supone un 0,1% del total. Si observamos mas detenidamente el gráfico de la figura 5.22 observamos también que no solo detenemos el método pronto sino que además el error real cometido en el punto donde se detiene es de un valor inferior al 1%.

Una vez hemos comprobado que el método funciona también para aplicaciones de SPEC00, vamos a comprobar si la mejora

aplicada anteriormente a GO, LI y M88K también funciona en este tipo de aplicaciones.

A B

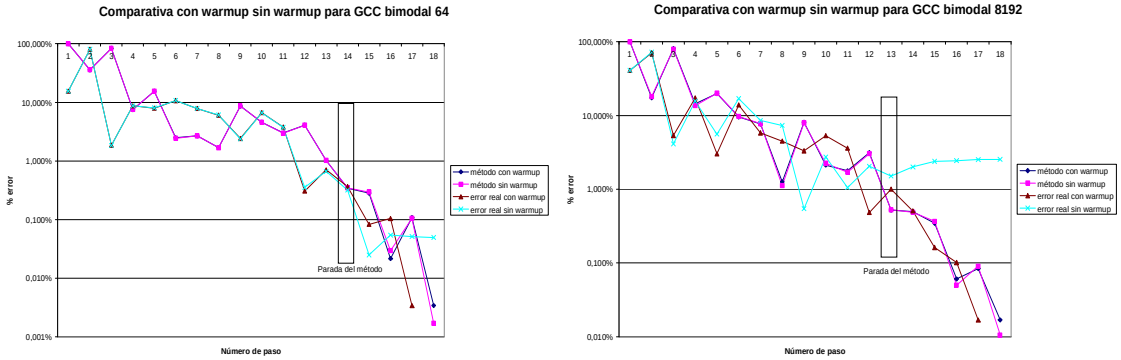


Figura 5.23-A: Gráfica que muestra el error cometido en cada paso de simulación con y sin warmup con configuración bimodal de 64 para GCC.
B: Gráfica que muestra el error cometido en cada paso de simulación con y sin warmup con configuración bimodal de 8192 para GCC.

La figura 5.23 muestra el error cometido por el método y el error real que se obtiene en cada paso de la aplicación GCC aplicándole un pequeño calentamiento inicial y sin aplicárselo. Recordemos que la aplicación GCC paraba el método en las configuraciones bimodal de 64 bits en el paso 14 y bimodal de 8192 en el paso 14. Como se puede comprobar en las gráficas de la figura 5.23, la detención del método es exactamente igual que la presentada anteriormente. No obstante se puede apreciar que el error real con warmup se ve disminuido respecto al error real antes obtenido.

A B

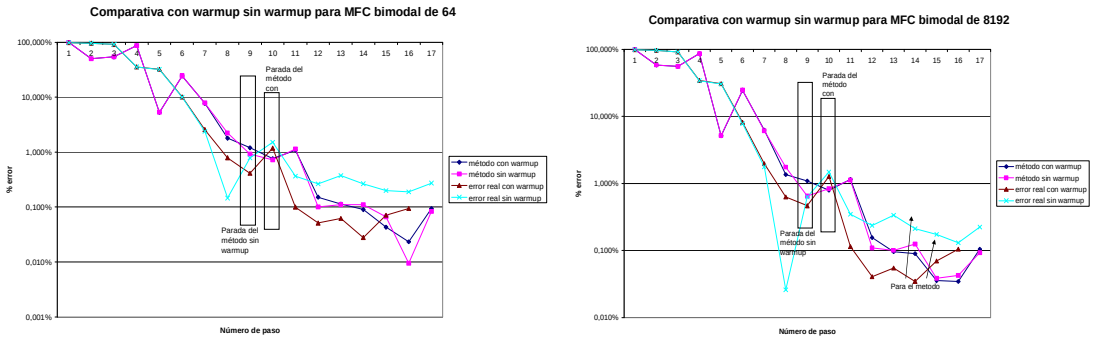


Figura 5.24-A: Gráfica que muestra el error cometido en cada paso de simulación con y sin warmup con configuración bimodal de 64 para Mcf.
B: Gráfica que muestra el error cometido en cada paso de simulación con y sin warmup con configuración bimodal de 8192 para Mcf.

La figura 5.24 muestra el error cometido por el método y el error real que se obtiene en cada paso de la aplicación Mcf aplicándole un pequeño calentamiento inicial y sin aplicárselo. Para la aplicación Mfc el método lo deteníamos en el paso 9 para ambas configuraciones. El hecho de aplicar un pequeño intervalo de iniciación no ha surtido efecto a la hora de aplicarlo al método. Según se puede apreciar en las gráficas de la figura 5.24, el error del método acostumbra a ser menor en la configuración si el intervalo inicial. La diferencia también se puede observar en el lugar donde detenemos el método. El método se detiene un paso más tarde que si no le hubiéramos aplicado el calentamiento. Pese a tener que detener el método un paso mas tarde que sin calentamiento, también se puede observar en las gráficas 5.24 que el error real es inferior en el caso de aplicar un intervalo inicial.

A B

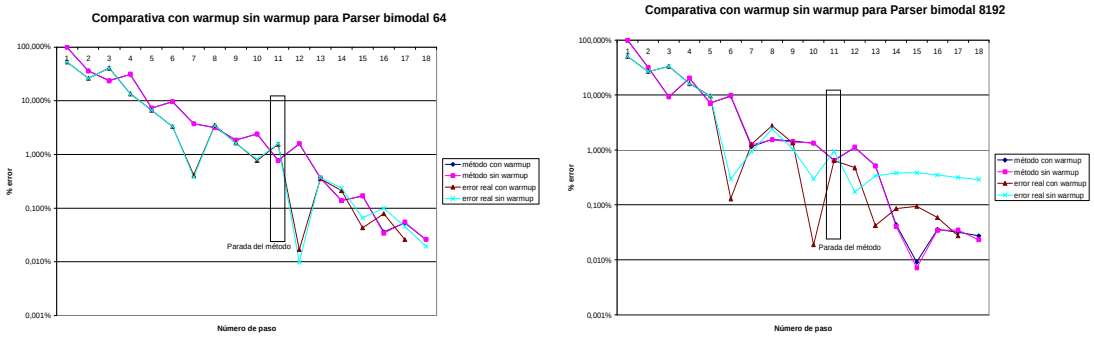


Figura 5.25-A: Gráfica que muestra el error cometido en cada paso de simulación con y sin warmup con configuración bimodal de 64 y 8192 para Parser.
B: Gráfica que muestra el error cometido en cada paso de simulación con y sin warmup con configuración bimodal de 64 y 8192 para Parser.

La figura 5.25 muestra el error cometido por el método y el error real que se obtiene en cada paso de la aplicación Parser aplicándole un pequeño calentamiento inicial y sin aplicárselo. La simulación del método para la aplicación Parser supuso una detención en el intervalo 11. Justo el mismo paso en el que se detiene aplicándole un warmup inicial. Se puede observar que las líneas que marcan el error cometido por el método van muy parejas, por lo que el error no dista mucho uno del otro. La mejora se observa cuando calculamos el error real de la aplicación. En la configuración bimodal de 8192 bits es

donde la diferencia se acentúa más, dado que a mayor tamaño de historia mayor es la mejora al aplicarle una inicialización.

A B

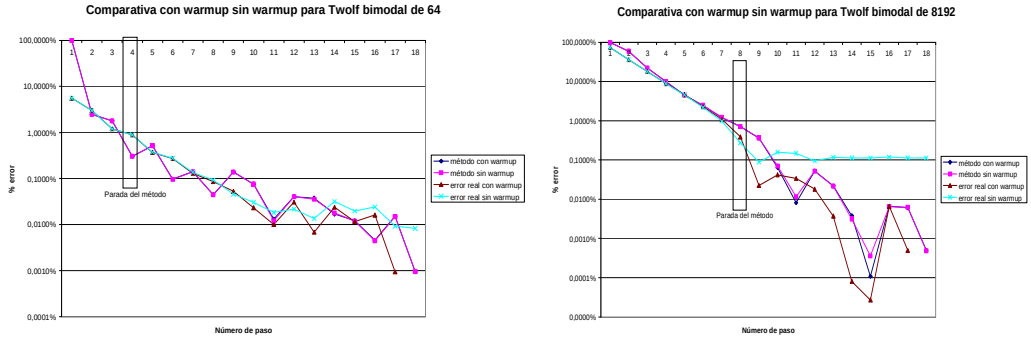


Figura 5.26-A: Gráfica que muestra el error cometido en cada paso de simulación con y sin warmup con configuración bimodal de 64 para Twolf.
B: Gráfica que muestra el error cometido en cada paso de simulación con y sin warmup con configuración bimodal de 8192 para Twolf.

La figura 5.26 muestra el error cometido por el método y el error real que se obtiene en cada paso de la aplicación Parser aplicándole un pequeño calentamiento inicial y sin aplicárselo. Como se puede observar en la figura 5.26 el error del método en los casos con y sin calentamiento inicial es prácticamente idéntico. Hay algunos determinados puntos en los que parece que el método con warmup obtiene un error menor, pero son solo pasos puntuales. Como se aprecia en ambas gráficas de la figura 5.26 también se detiene el método en los mismos puntos donde deteníamos sin aplicar un calentamiento inicial. Las diferencias de error real en el esta aplicación son bastante más visibles. Parece que aplicarle la mejora surte efecto, puesto que sobretodo en la gráfica 5.26-B donde se observa una mejora abismal sobretodo en la parte final de la simulación.

5.5.2. Con warmup

Con los datos mostrados gráficamente del lugar de la detención del método en cada uno de los casos vamos a ver más gráficamente la diferencia entre ellos para hacernos mas a la idea de cuanto supone la mejora. En el caso de la aplicación MFC mostraremos dos

resultados, el error en el momento en que el método detiene la simulación sin calentamiento y el lugar donde el método detiene la simulación con calentamiento. También decir que el error real de las gráficas con warmup termina un paso antes puesto que están mostradas en escala logarítmica y el error de la simulación completa con warmup respecto del real es 0.

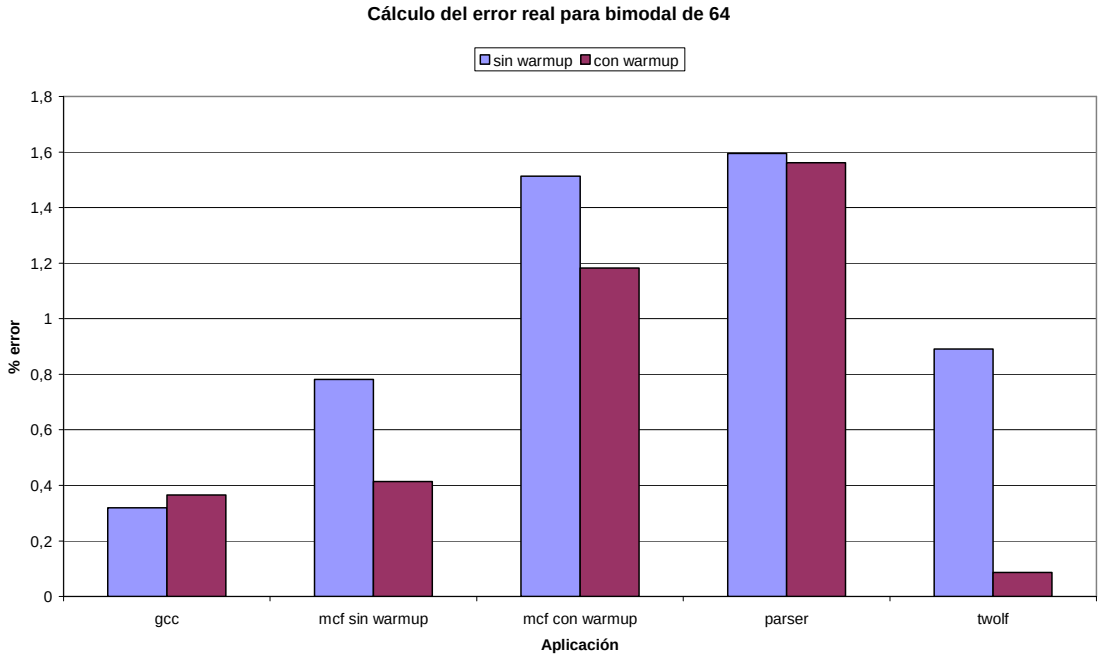


Figura 5.27: Gráfica que muestra el error real cometido por cada aplicación con y sin warmup para la configuración del predictor bimodal de 64Bits.

Como se puede comprobar en la gráfica 5.27 utilizar un pequeño calentamiento antes de simular el intervalo hace que se produzca menos error respecto a la simulación completa de la aplicación. La aplicación "mcf sin warmup" muestra el error cometido en el punto de detención si no se le aplica un calentamiento. La aplicación "mcf con warmup" es el error real cometido por ambos casos en el punto que detenemos el método con warmup. Exceptuando la aplicación GCC en todas las demás se observa una mejora del error real. Donde mas se observa esa mejora es en la aplicación Twolf.

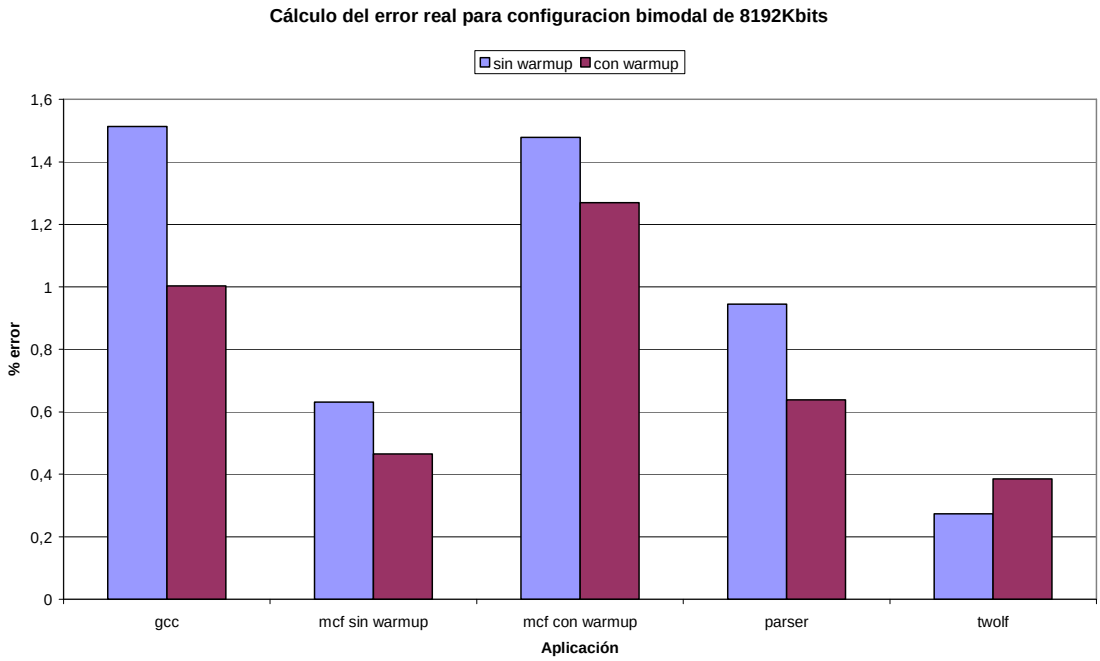


Figura 5.28: Gráfica que muestra el error real cometido por cada aplicación con y sin warmup para la configuración del predictor bimodal de 8192Bits.

Observando la grafica de la figura 5.28, podemos comprobar que aplicar un calentamiento inicial produce una mejora en el error real respecto a no aplicárselo. Cabe destacar que pese a ver que la aplicación Twolf ofrece un error mayor con warmup que si él, se puede decir que es el único punto en toda la grafica en el que se da este caso. Tanto antes de la detención de la simulación en menor diferencia, como después el aplicar un warmup al inicio de cada intervalo supone una mejora sustancial.

5.6. Conclusiones obtenidas

El método incremental utilizado en el estudio expuesto muestra que se puede reducir el tiempo de simulación haciendo que únicamente tengamos que simular entre un 1% y un 5% de la aplicación para obtener un resultado fiable. El resultado obtenido simulando un porcentaje tan bajo difiere del resultado real entre un 1% y un 6% aproximadamente. Estos resultados dan a entender el buen funcionamiento del método. Pese a ver que los resultados eran favorables y comprobar que las hipótesis iniciales se cumplían, se

decidió aplicar una pequeña mejora, con la esperanza que los resultados mejoraran aun más.

La mejora implica el tener que simular un pequeño intervalo extra, con lo que el tiempo de simulación se vería aumentado levemente. Numéricamente se observa que el resultado obtenido por la mejora reduce el error a valores de entre una décima parte y una centésima parte. Estos datos demuestran el buen comportamiento del método y los resultados mejorados por la variación realizada.

Los resultados obtenidos por el warmup dan a entender también que siempre que se simulen los mismos pasos con el método utilizado el resultado se ve mejorado. La estimación del error se podría mejorar seguramente si se aplicara otro método de elección de los intervalos que no fuera regular. El método empleado en el estudio escoge los intervalos a distancias reculares, esto puede llevar escoger los intervalos desfavorables del comportamiento de la aplicación por lo que el resultado que se obtendría seria muy alejado del real. El utilizar un método más aleatorio en la elección del intervalo a simular podría dar pie a no coincidir en el comportamiento de la aplicación. Para realizar este tipo de elección aleatoria de intervalos se debería tener en cuenta el comportamiento de la aplicación y comprobar las zonas diferenciadas de simulación. De cada zona se deberán escoger los intervalos de forma equitativa para que el resultado fuera óptimo.

6. Conclusiones y opinión personal

En este proyecto se han realizado las fases necesarias para la realización de un estudio. Primeramente se ha tenido que buscar la documentación necesaria para entender el problema. Esta documentación constaba de términos científicos y manuales para entender el funcionamiento del simulador.

Una vez entendido el problema se procede a la familiarización con el entorno de trabajo. En nuestro caso el entorno de trabajo es el simulador Simpoint y tres aplicaciones de prueba de SPEC95. Después de realizar una serie de pruebas para entender el funcionamiento del simulador se decide modificar el código para poder obtener resultados más rápidamente. Al obtener los datos de las primeras simulaciones éstos son procesados en hojas de Excel y mostrados en gráficas para analizar más rápidamente los resultados.

Cuando se tiene el entorno preparado para la realización del estudio se buscan soluciones al problema. Se exponen unas cuantas soluciones como propuestas. Se escoge una propuesta como posible solución al problema y se empiezan los experimentos. Nuevamente los datos son procesados en Excel y graficados para mejor comprensión y análisis. Se puede comprobar que el método escogido es un método que funciona y se logra reducir el tiempo de simulación notablemente. A la vez que se logra reducir el tiempo de simulación comprobamos que el error que se produce está dentro del baremo escogido como bueno. Además se decide aplicar una mejora al método inicial para mejorar el resultado final. Como se esperaba en un principio, la mejora aplicada al método escogido muestra unos resultados satisfactorios. Podemos concluir que el método escogido para la realización del estudio.

Pese a haber obtenido unos resultados más que satisfactorios en el estudio cabe añadir que estos resultados pueden mejorar

realizando otro tipo de mejoras. Una de estas mejoras propuesta es la de utilizar el mismo método con programación en paralelo.

El proyecto me ha parecido desde el primer día interesante. Mi idea de hacer un proyecto de final de carrera no era la de hacer un programa que calculara un valor o una Web para registrar algo. La intención que tenia desde el principio era la de realizar un trabajo de laboratorio mostrando posteriormente los resultados obtenidos de una investigación.

7. Líneas abiertas

Viendo que el método funciona perfectamente para cualquier aplicación, vamos a exponer una serie de posibles mejoras para obtener un resultado en un tiempo menor y si cabe mejor que el obtenido hasta el momento. Los resultados de estas mejoras no están demostrados científicamente, únicamente son hipótesis iniciales y estimaciones de mejora respecto al estudio expuesto.

7.1. Programación en paralelo

Una posible mejora consistiría en tener una red de ordenadores conectados. Cada ordenador debería lanzar la simulación de uno de los intervalos y el resultado pasarlo a un ordenador centralizado.

Tal y como muestra la figura 5.29 el ordenador central dividiría el trabajo entre el numero de maquinas para hacerlo lo mas equitativamente posible. Naturalmente una parte indivisible es un intervalo por lo que si tenemos 5 intervalos a simular se lanzan en 5 maquinas diferentes, aunque se tengan 10 disponibles.

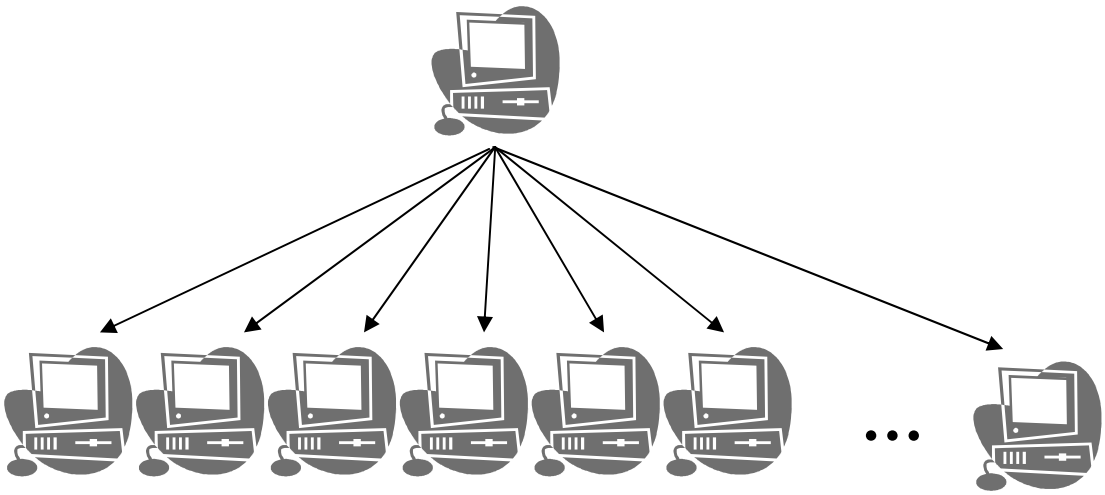


Figura 5.29: Ejemplo de cómo se realizaría una simulación con varios procesadores.

Para la realización del método habría principalmente dos posibilidades. Una primera posibilidad seria aplicar el método desde la maquina principal y mandando a simular los intervalos que correspondan en cada paso a las maquinas disponibles. Esto daría el

mismo resultado que en monoprocesador pero mejoraría en el tiempo. Otra posibilidad seria dividir la aplicación entre el número de procesadores disponibles. Enviar a cada procesador una parte de la aplicación. Cada procesador debe aplicar el método al fragmento de aplicación que le ha tocado. El resultado se devolvería al procesador central donde únicamente se debería hacer un promedio para obtener el resultado. De esta forma el error debería ser mucho menor dado que el número de intervalos simulados globalmente debería ser mayor.

Si aun se pretende mejorar el resultado con este método, se puede aplicar la mejora antes expuesta de un pequeño calentamiento inicial antes de cada intervalo. Ya se ha demostrado anteriormente que aplicando un calentamiento inicial al intervalo para llenar la historia del predictor de saltos se obtiene un error mas bajo que sin aplicarlo.

7.2. Mejora en la estimación del error

Una posible mejora que se podría realizar al estudio presentado es la de mejorar la estimación del error durante la simulación. Para ello se podrían tener en cuenta los N pasos anteriores en vez de contar únicamente con el paso anterior. Se podría considerar el hecho de calcular dentro de cada intervalo el comportamiento mediante más subintervalos. Se podrían usar métodos estadísticos complejos para estimar el error.

También se podría considerar el caso de no escoger los intervalos a distancia equidistante y hacerlo de forma más aleatoria. Con ello evitaríamos coincidir la elección de los intervalos simulados con un comportamiento desfavorable y podría dar pie a un resultado con menos fluctuaciones en cada paso.

8. Bibliografía

Libros

Título: Arquitectura de computadores 4ª edición
Autores: John Paul shen, Mikko. H Liposti
Editorial Mc Graw Hill

Título: Organización y arquitectura de computadores
Autores: William Stallings
Editorial Prentice Hall

Revistas

Título del artículo: Analisys of Simulation-adapted SPEC 2000 Benchmarks
Autores: Gómez, L. Piñuel, M. Prieto, F. Tirado
Revista: ACM SIGARCH Computer Architecture News
Páginas: 4 - 10
Año: 2002

Título del artículo: Phase Tracking and Prediction
Autores: Timothy Sherwood, Suleyman Sair, Brad Calder
Revista: Computer Society
Páginas: 1 - 12
Año: 2003

Título del artículo: Automatically Characterizing Large Scale Program Behaivior
Autores: Timothy Sherwood, Erez Perelman, Greg Hamerly, Brad Calder
Revista: Architectural Support for Programming Languages and Operating Systems
Páginas: 45 - 57
Año: 2002

Título del artículo: Basic Block Distribution Analisys to Find Periodic Behaivior and Simulation Points in Applications
Autores: Timothy Sherwood, Erez Perelman, Brad Calder
Revista: PACT, Proceedings of the internacional conference on Parallel Architectures and compilation techniques
Páginas: 3 - 14
Año: 2001

Paginas Web

http://www.simplescalar.com/
Web oficial del simulador simple scalar utilizado para realizar el proyecto.

http://ieeexplore.ieee.org/Xplore/dynhome.jsp
Buscador de referencias de artículos sobre temas informáticos.

http://ieeexplore.ieee.org/xpl/conferences.jsp
Web conferencias sobre predictores de saltos.

http://portal.acm.org/dl.cfm
Micro arquitectura.

http://portal.acm.org/browse_dl.cfm?linked=1∂=series&idx=SERIES311&coll=ACM&dl=ACM&CFID=50564683&CFTOKEN=75730635
Web relacionada con los sistemas operativos y su funcionamiento.

http://www.spec.org/
Web relacionada con los benchmark.

Resumen

En este proyecto se pretende realizar un estudio para comprobar si se puede simular una aplicación de prueba (benchmark) reduciendo el tiempo de simulación. Para reducir el tiempo de simulación se seleccionarán ciertos fragmentos significativos de la ejecución de la aplicación. El objetivo es obtener un resultado de simulación lo más similar posible a la simulación completa pero en menos tiempo. Para ello usamos un método que llamamos incremental y que consiste en dividir la simulación en intervalos de un millón de instrucciones. La simulación consiste en una serie de pasos. En cada paso se van añadiendo intervalos y la simulación se detiene cuando la diferencia entre el resultado del paso actual y el del anterior es inferior a un valor escogido al inicio. Posteriormente se propone una mejora del método y se muestran los resultados. La mejora consiste en simular un pequeño intervalo previo al intervalo significativo para mejorar así el resultado.

Resum

Aquest projecte tracta de realitzar un estudi per comprovar si es pot simular una aplicació de prova (benchmark) reduint el temps de simulació. Per tal de reduir el temps de simulació es seleccionaran uns determinats fragments significatius de l'execució de l'aplicació. L'objectiu es obtenir un resultat de simulació el més similar possible al de la simulació completa però en menys temps. El mètode que farem servir s'anomena incremental y consisteix a dividir la simulació en intervals d'un milió d'instruccions. Un cop dividit em simular per passos. En cada pas es van afegint intervals i s'atura la simulació quan la diferencia entre el resultat del pas actual i l'anterior es inferior a un determinat valor escollit inicialment. Després es proposa una millora que es realitza i es mostren els resultats obtinguts. La millora consisteix en simular un petit interval previ a l'interval significatiu per millorar el resultat.

Summary

This Project consists in a study to proof whether is possible to simulate a benchmark reducing the simulation time. In order to reduce the simulation time we choose the significant fragments for being simulated. The objective is to obtain a similar result as if it was a complete simulation. The used method is called increasing. The method consists in dividing the simulation in a million intervals. The method consists in making passes. Each pass we have to increase the number of intervals we simulate. We stop when the difference between the result of the actual pass and the behind one is less than a value chosen initially. Furthermore we have proposed an improvement which is done and the results shown. The improvement consists in applying a little warm up before we simulate each interval to obtain a better result.