UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Evaluación experimental de los mecanismos de prebúsqueda en el IBM Power8

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Autor: Jahel Carmona Vila
Tutor: Julio Sahuquillo Borrás
Director Experimental: Josué Feliu Pérez

Curso 2017-2018

Resum

En aquest projecte s'estudien els mecanismes de precerca en una màquina amb el processador IBM Power8. S'executen aplicacions SPEC i s'estudia l'impacte sobre les prestacions que exerceixen els diferents mecanismes de precerca que implementa la màquina. L'estudi caracteritza el comportament de les aplicacions i les classifica segons els beneficis obtinguts en les prestacions. El processador implementa una gran varietat de "prefetchers" configurables mitjançant l'accés al registre adient. S'avalua l'efectivitat dels mateixos treballant de forma individual.

Paraules clau: Precerca, latències de memòria, màquines reals

Resumen

En este proyecto se estudian los mecanismos de prebúsqueda en una máquina con el procesador IBM Power8. Se ejecutan benchmarks SPEC y se estudia el impacto sobre las prestaciones que ejercen los distintos mecanismos de prebúsqueda que implementa la máquina. El estudio caracteriza el comportamiento de las aplicaciones clasificándolas según los beneficios obtenidos en las prestaciones. El procesador implementa una gran variedad de "prefetchers" configurables mediante el acceso al registro adecuado. Se evalua la efectividad de los mismos trabajando de manera individual.

Palabras clave: Prebúsqueda, latencias de memoria, máquinas reales

Abstract

In this project are studied the prefetch mechanisms on a machine with the IBM Power8 processor. SPEC benchmarks are executed and the impact on the performance of the different prefetch mechanisms implemented by the machine are studied. The study characterizes the behavior of applications by classifying them according to the benefits obtained in the performance. The processor implements a wide variety of configurable "prefetchers" by accessing the appropriate register. It is evaluated its effectiveness working individually.

Key words: Prefetch, memory latency, real machines

Índice general

Índice de figuras

Índice de tablas

Agradecimientos

A Julio, Josué y Carlos por ayudarme, corregirme y ser tan críticos siempre. Es así como se consigue descubrir.

A mis padres, que siempre apostaron por aprender, y aquí estoy gracias a ellos.

A David, por creer siempre en mi.

CAPÍTULO 1
Introducción

Motivación

Los *IBM Power8* son una familia de procesadores superescalares simétricos, basados en la arquitectura *Power*. Los primeros sistemas basados en esta arquitectura diseñada por IBM aparecieron a partir de Junio de 2014; aparecieron también los primeros diseños de miembros de la fundación *OpenPOWER* —organización que concede licencias a aquellos que quieran implementar la arquitectura *Power*— a principios de 2015. Han habido otras familias de *Power*, siendo estas desde la 1 hasta la actual, y en 2016 IBM anuncia la familia *Power9*, con otro tipo de mejoras respecto a la que nos ocupa.

Los procesadores *Power8* tienen la peculiaridad de ser capaces de manejar ocho hilos de ejecución por cada núcleo. Este procesador puede incluir desde cuatro hasta doce núcleos, y cada núcleo hasta 8 hilos, por lo que se puede observar la cantidad masiva de hilos que se manejan. La máquina con la que se ha trabajado en el presente proyecto dispone de diez núcleos, por lo que tenemos ochenta hilos de ejecución lógicos.

Respecto al sistema de memoria, el procesador implementa cuatro niveles de caché, ubicando los tres primeros dentro del chip y el cuarto fuera, característica por la cual no será objeto de nuestro estudio. El resto de cachés son:

- Caché Nivel 1 (L1 Cache): Separada para datos e instrucciones. La caché de datos tiene una capacidad de almacenamiento de 64KB y la caché de instrucciones de 32KB.
- Caché Nivel 2 (L2 Cache): Compartida para datos e instrucciones pero individual para cada núcleo, implementada con tecnología SRAM. TIene un tamaño de 512KB de capacidad por cada núcleo, es decir, en nuestro caso la capacidad total se multiplica por diez, lo que resulta en un total de 5MB.
- Caché Nivel 3 (L3 Cache): Compartida para datos e instrucciones, implementada con tecnología eDRAM. Está dentro del chip y es compartida para todos los núcleos y dispone de una capacidad de 96MB.

La *prebúsqueda en caché*, o **cache prefetching**, en inglés, como es más común, es una técnica utilizada por los procesadores para acelerar el rendimiento, a través de la búsqueda de datos y/o instrucciones en una localización de memoria inferior y más lenta hacia una memoria superior en la jerarquia y más rápida. Esto es porque las cachés son más rápidas cuando más pequeñas son, y la memoria principal tiene un tiempo de acceso

mucho mayor. Así, se prebusca el dato en memoria principal y se trae a la L1, la caché más pequeña. Con prebúsqueda indicamos que la búsqueda se realiza antes que el procesador la solicite. Esto se realiza gracias a que el *hardware* identifica patrones de acceso y, cuando detecta un patrón, activa un mecanismo (*prefetch prefetcher*) para "acercar" los datos al procesador. De esta manera, cuando el procesador realice la búsqueda de los datos, ésta se hace antes, reduciendo el tiempo de acceso a los datos y, por ende, el tiempo de ejecución.

El nodo que realiza la prebúsqueda tiene un registro de bits llamado DSCR, *Data Streams Control Register*. Está compuesto por 64 bits, que se agrupan en una serie de subregistros para controlar diferentes parámetros del prefetch. Cada subregistro controla una característica de la prebúsqueda, y son exactamente los registros que hay a continuación:

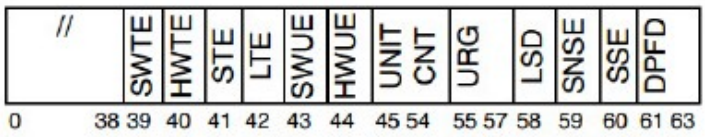

Figura 1.1: Registro DSCR en el procesador IBM Power8

Los primeros 38 bits de la imagen, que no aparecen con ningún nombre, no tienen que ver con la parametrización de la prebúsqueda, motivo por el cual no se han mostrado. Los demás bits sí, cuyo significado explicamos a continuación:

- SWTE [bit 39]: *Software Transient Enable*, Habilitación transitoria de flujos definidos a nivel software.
- HWTE [bit 40]: *Hardware Transient Enable*, Habilitación transitoria de detección de flujos a nivel hardware.
- STE [bit 41]: *Store Transient Enable*, Habilitación transitoria de flujos almacenaje.
- LTE [bit 42]: *Load Transient Enable*, Habilitación de carga transitoria
- SWUE [bit 43]: *Software Unit count Enable*, Habilitación de la cuenta de unidades a los flujos de software definidos.
- HWUE [bit 44]: *Hardware Unit count Enable*, Habilitación de la cuenta de unidades a los flujos de hardware detectados.
- UNITCNT [bits 45-54]: *Unit Count*, Número de unidades en el flujo de datos.
- URG [bits 55-57]: *Depth Attainment Urgency*, Urgencia con la que el prefetch alcanza la profundidad configurada por los flujos de hardware detectados. Este conjunto de bits tiene ocho opciones, que significan 0 urgencia por defecto, 1 no hay urgencia y 7 el más urgente.
- LSD [bit 58]: *Load Stream Disable*, Deshabilita la detección de harware y la iniciación de los flujos de cargas de datos.
- SNSE [bit 59]: *Stride-N Stream Enable*, Habilitación de los flujos de detección de hardware e iniciación de carga y almacenaje de datos que tienen un alcance, es decir, un *stride* mayor que un bloque de caché. Los flujos de carga se detectan cuando LSD = 0 y los flujos de almacenaje cuando SSE = 1.

- SSE [bit 60]: *Store Stream Enable*, Habilitación de los flujos de detección de hardware e iniciación de almacenaje.
- DPFD [bits 61-63]: *Default Prefetch Depth*, Profundidad del prefetch que pueden alcanzar los flujos de hardware detectados, es decir. Este conjunto de bits tiene ocho opciones, que significan 0 profundidad por defecto, 1 no hay profundidad (por lo tanto, no hay prefetch) y 7 el más profundo, es decir, el que más datos obtiene.

Los grupos de bits más significativos y que hemos escogido para nuestro estudio son URG y DPFD. Esto es debido a que con ellos, podemos controlar la cantidad de bloques en memoria que se prebuscan, es decir, si se toman pocos o muchos bloques a cada orden de prebúsqueda, y la urgencia, o prioridad, con la que éstos se prebuscan. Esto es determinante en muchos benchmarks, como veremos más adelante.

La estructura del prefetch a nivel físico es un tema ya tratado; se conocen los registros por los cuales está compuesto, la función de cada registro y demás aspectos típicos de estructuras físicas. El interés del tema radica en que no se ha determinado en ningún estudio, o almenos de manera experimental, cómo afecta el caché prefetching al rendimiento de la máquina, el cual es relevante para aquellas aplicaciones en las que se desee obtener un rendimiento determinado mediante el ajuste de los parámetros de la prebúsqueda en el subsistema de memoria.

Objetivos

El objetivo de este proyecto es justificar el rendimiento que se obtiene de las ejecuciones de las aplicaciones variando las diferentes configuraciones que se pueden hacer con el registro de la prebúsqueda. Para ello se utilizarán los contadores hardware, que medirán las variables que hacen falta para aplicar cálculos que cuantifiquen las prestaciones del sistema.

A su vez, estos calculos los vamos a comparar entre ellos y estableceremos cual es la mejor configuración para según qué aplicación. Queremos que en un futuro este estudio sirva de base y se utilice para que se pueda construir un afinador del prefetcher, que seleccione la configuración que proporcione mejores prestaciones en función de las necesidades de la aplicación.

Para ello, hemos utilizado los contadores hardware accesibles por las herramientas *Perf* y *Oprofile*, pues los segundos permiten utilizar contadores que no tienen los primeros. Estos contadores los hemos introducido a través de un programa en C, utilizando librerías de *Perf*. Para configurar el prefetcher, hemos empleado una orden, *ppc64_cpu* que nos brinda Linux, y la hemos aplicado a través de Shell Script.

Estructura de la memoria

La memoria la hemos estructurado en tres partes. En la primera, el capítulo 2, definimos las herramientas que hemos empleado para el estudio; explicamos los códigos que hemos empleado, las aplicaciones para el testeo, los contadores de eventos utilizados y las herramientas para obtener los resultados. En la segunda, el capítulo 3, contamos el desarrollo del estudio, es decir, se presenta un estudio del rendimiento de las aplicaciones, separando éstas en coma flotante y enteros —ya que tienen rendimientos diferentes

y tratarlas juntas puede ocultar observaciones interesantes—, y buscando razonamientos que justifiquen el comportamiento en las prestaciones de las aplicaciones. En la tercera y última parte, capítulo 4, se presentan los resultados de un experimento controlado para corroborar ciertas hipótesis sobre el comportamiento irregular de algunas aplicaciones y confirmar los resultados que esperamos según las hipótesis planteadas.

CAPÍTULO 2
Herramientas

Para el estudio hemos empleado una serie de herramientas. Hemos utilizado un script escrito en Shell Script, que lanza un programa en C, el cual contiene las métricas de prestaciones que queremos medir durante la ejecución de los benchmarks de prueba. El script genera un fichero que más tarde analizamos y convertimos a formato CSV para poder tratarlo como una hoja de cálculo, para poder efectuar operaciones sobre los datos obtenidos. Con más detalle, procedemos a explicarlo en las siguientes secciones.

Configuración del prefetcher

A la hora de establecer los bits en el *Data Stream Control Register*, o DSCR, necesitamos una macro que actúe como máscara. Esta macro existe en el conjunto de órdenes de Linux bajo el nombre de **ppc64_cpu**. Se tiene que ejecutar como super usuario para poder tener permisos sobre ella. Dentro de los parámetros de los que la orden dispone, se ha empleado solamente el *–dscr*. Su uso es el siguiente:

```
1 ppc64_cpu --dscr                    # Obtiene la configuracion actual del DSCR
2 ppc64_cpu --dscr=<val>              # Cambia los bits del DSCR por <val>
3 ppc64_cpu --dscr [-p <pid>]         # Obtiene la configuracion actual del DSCR
                                        por proceso <pid>
4 ppc64_cpu --dscr=<val> [-p <pid>] # Cambia los bits del DSCR por <val> por
                                        proceso <pid>
```

Listing 2.1: Script en Shell Script

Las opciones de cambio de valor del registro por cada proceso no las hemos empleado, pero en un futuro se pueden considerar para desarrollos que se pueden obtener a partir de este estudio, pues el procesador tiene un gran potencial en materia de manejo de procesos e hilos.

Spec CPU2006

El SPEC CPU2006, o *Standard Performance Evaluation Corporation benchmark suite*, es un conjunto de programas que realizan un consumo intenso, estandarizados para industria, que estresan el procesador del sistema, el subsistema de memoria y el compilador. Existen dos grupos de SPEC dentro de este paquete, que se diferencian en los SPECint® y los SPECfp®, siendo los primeros programas que realizan numerosas aplicaciones aritméticas con números enteros y los segundos programas de números de coma flotante.

El objetivo de este paquete de aplicaciones es obtener una medida comparativa de la intensidad de computación en el amplio rango de opciones del hardware, utilizando como base de la aplicación cargas de trabajo desarrolladas por usuarios representativos.

Las aplicaciones del paquete CPU2006 que hemos empleado son, las de números enteros: **Bzip2, Gcc, Mcf, Gobmk, Hmmer, Sjeng, Libquantum, H264ref, Omnetpp, Astar** y **Xalancbmk**. Las aplicaciones de coma flotante son **Bwaves, Gamess, Milc, Zeusmp, Gromacs, CactusAMD, Leslie3D, Namd, Soplex, Povray, GemsFDTD** y **Lbm**.

Shell Script

Como lanzador del programa en C, que es el que realmente mide los eventos, tenemos este script en Shell Script. Debido a que la única manera de configurar el prefetcher es a través de una orden en Shell Script, lo que hace este conjunto de órdenes es configurar el DSCR con los valores especificados en un bucle (haciendo *echo* al terminal, lo ejecuta), selecciona un benchmark de la lista codificada de ellos y finalmente ejecuta el programa en C que hará la medición de los eventos que tiene configurados para el benchmark escogido. El listing 2.2 muestra este script. Los valores que introducimos en el DSCR en este script se explican en el listing 3.1.

```
1  for valor in 0 66 455 71 450 1;do
2    frase='ppc64_cpu --dscr=$valor'
3    echo $frase
4    for benchmark in 1 2 3 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 21 22 23; do
5      echo "#" $benchmark "=" $valor >> "num_instr_y_ciclos.dat"
6      { ./libpfm-4.7.0/perf_examples/executa_instruccions_120 -A $benchmark -d
           200; } 2>> "num_instr_y_ciclos.dat"
7    done
8  done
```

Listing 2.2: Script en Shell Script

Como podemos observar, la salida de error se escribe en *num_instr_y_ciclos.dat*. Esto es porque los benchmarks de CPU2006 utilizan la salida estándar para mostrar datos relativos a la aplicación, como el nivel de compresión de un dato o la iteración por la que va el programa; datos que a nosotros no nos interesan porque solamente queremos obtener los eventos medidos. Por ejemplo, tras la ejecución de un script se obtienen resultados similares a los presentados en el listing 2.3.

```
1  ...
2  # 3 = 0
3  PMU_COUNTS:          452752986598          186487927348          5763114007
                 8097198793          34317666705
4  # 5 = 0
5  PMU_COUNTS:          442998953769          776590082004          583025465
                   31896771            776548463
6  # 6 = 0
7  PMU_COUNTS:          443786435732          615478349923          276845917
                  150948503           1742465595
8  # 7 = 0
9  PMU_COUNTS:          446819005598          480542443101            5300326
                  594598171           8100175465
10 ...
```

Listing 2.3: Resultado de la ejecución del Script

La expresión que va entre la almohadilla y el símbolo de igual indica el número del benchmark ejecutado, lo que va a la derecha del igual es la configuración de prefetch con

la que se está ejecutando el benchmark, y lo que hay a continuación de *PMU_COUNTS* son los valores de los eventos medidos. Para saber a qué evento hace referencia cada cifra, hay que mirar los eventos que se le han pasado al programa en C, ya que está indicada la secuencia que se sigue en los eventos. Estos pueden pasarse o bien al invocar el programa con el parámetro "-e" o bien directamente escritos en el programa en C, explicado en el listing 2.4.

Programa en C a bajo nivel

Para hacer la medición de los eventos, empleamos un programa en C. Este es bastante extenso, pero a continuación se presenta una muestra de éste para a explicar las tareas realizadas en las principales partes del programa.

```
/*
count_instructions_ref.c
counts the number of instructions that a process executes during a period - uses reference inputs
./executa_instruccions_120 -A benckmark -d 200
*/

#include <sys/types.h>
#include <inttypes.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <errno.h>
#include <unistd.h>
#include <string.h>
#include <stdarg.h>
#include <sys/wait.h>
#include <err.h>
#include <sys/poll.h>
#include <sched.h>
#include "perf_util.h"

typedef struct {
  char *events;
  int delay;
  int pinned;
  int group;
  int verbose;
} options_t;

static options_t options;

//uint64_t cycles[7];
//uint64_t insts[7];

uint64_t pmu_counters[7];
char *events[7];

char *benchmarks[][200] = {
// 0 -> perlbench
{NULL, NULL, NULL},
// 1 -> bzip2
{"/home/jacarvi/working_dir/spec_bin/bzip2.ppc64", "/home/jacarvi/working_dir/CPU2006/401.bzip2/data/all/input/input.combined", "200", NULL},
// 2 -> gcc
{"/home/jacarvi/working_dir/spec_bin/gcc.ppc64", "/home/jacarvi/working_dir/CPU2006/403.gcc/data/ref/input/scilab.i", "-o", "scilab.s", NULL},
// 3 -> mcf
```

```
{"/home/jacarvi/working_dir/spec_bin/mcf.ppc64", "/home/jacarvi/working_dir/
    CPU2006/429.mcf/data/ref/input/inp.in", NULL},
// 4 -> gobmk
{"/home/jacarvi/working_dir/spec_bin/gobmk.ppc64", "--quiet", "--mode", "gtp",
    NULL},
// 5 -> hmmer
{"/home/jacarvi/working_dir/spec_bin/hmmer.ppc64", "--fixed", "0", "--mean", "
    500", "--num", "500000", "--sd", "350", "--seed", "0", "/home/jacarvi/
    working_dir/CPU2006/456.hmmer/data/ref/input/retro.hmm", NULL},
// 6 -> sjeng
{"/home/jacarvi/working_dir/spec_bin/sjeng.ppc64", "/home/jacarvi/working_dir/
    CPU2006/458.sjeng/data/ref/input/ref.txt", NULL},
// 7 -> libquantum
{"/home/jacarvi/working_dir/spec_bin/libquantum.ppc64", "1397", "8", NULL},
// 8 -> h264ref
{"/home/jacarvi/working_dir/spec_bin/h264ref.ppc64", "-d", "/home/jacarvi/
    working_dir/CPU2006/464.h264ref/data/ref/input/foreman_ref_encoder_baseline
    .cfg", NULL},
// 9 -> omnetpp
{"/home/jacarvi/working_dir/spec_bin/omnetpp.ppc64", "/home/jacarvi/working_dir
    /CPU2006/471.omnetpp/data/ref/input/omnetpp.ini", NULL},
// 10 -> astar
{"/home/jacarvi/working_dir/spec_bin/astar.ppc64", "/home/jacarvi/working_dir/
    CPU2006/473.astar/data/ref/input/BigLakes2048.cfg", NULL},
// 11 -> xalancbmk
{"/home/jacarvi/working_dir/spec_bin/Xalan.ppc64", "-v", "/home/jacarvi/
    working_dir/CPU2006/483.xalancbmk/data/ref/input/t5.xml", "/home/jacarvi/
    working_dir/CPU2006/483.xalancbmk/data/ref/input/xalanc.xsl", NULL},
// 12 -> bwaves
{"/home/jacarvi/working_dir/spec_bin/bwaves.ppc64", NULL},
// 13 -> gamess
{"/home/jacarvi/working_dir/spec_bin/gamess.ppc64", NULL},
// 14 -> milc
{"/home/jacarvi/working_dir/spec_bin/milc.ppc64", NULL},
// 15 -> zeusmp
{"/home/jacarvi/working_dir/spec_bin/zeusmp.ppc64", NULL},
// 16 -> gromacs
{"/home/jacarvi/working_dir/spec_bin/gromacs.ppc64", "-silent", "-deffnm", "/
    home/jacarvi/working_dir/CPU2006/435.gromacs/data/ref/input/gromacs", "-
    nice", "0", NULL},
// 17 -> cactusADM
{"/home/jacarvi/working_dir/spec_bin/cactusADM.ppc64", "/home/jacarvi/
    working_dir/CPU2006/436.cactusADM/data/ref/input/benchADM.par", NULL},
// 18 -> leslie3d
{"/home/jacarvi/working_dir/spec_bin/leslie3d.ppc64", NULL},
// 19 -> namd
{"/home/jacarvi/working_dir/spec_bin/namd.ppc64", "--input", "/home/jacarvi/
    working_dir/CPU2006/444.namd/data/all/input/namd.input", "--iterations", "
    38", "--output", "namd.out", NULL},
// 20 -> dealII
{NULL, NULL, NULL},
// 21 -> soplex
{"/home/jacarvi/working_dir/spec_bin/soplex.ppc64", "-s1", "-e","-m45000", "/
    home/jacarvi/working_dir/CPU2006/450.soplex/data/ref/input/pds-50.mps",
    NULL},
//22 -> povray
{"/home/jacarvi/working_dir/spec_bin/povray.ppc64", "/home/jacarvi/working_dir/
    CPU2006/453.povray/data/ref/input/SPEC-benchmark-ref.ini", NULL},
// 23 -> GemsFDTD
{"/home/jacarvi/working_dir/spec_bin/GemsFDTD.ppc64", NULL},
// 24 -> lbm
{"/home/jacarvi/working_dir/spec_bin/lbm.ppc64", "300", "reference.dat", "0", "
    1", "/home/jacarvi/working_dir/CPU2006/470.lbm/data/ref/input/100
    _100_130_ldc.of", NULL},
```

```
89 // 25 -> tonto
90 {"/home/jacarvi/working_dir/spec_bin/tonto.ppc64", NULL},
91 // 26 -> calculix
92 {"/home/jacarvi/working_dir/spec_bin/calculix.ppc64", "-i", "/home/jacarvi/
       working_dir/CPU2006/454.calculix/data/ref/input/hyperviscoplastic", NULL},
93 // 27
94 {NULL, NULL, NULL},
95 };
96
97 unsigned long int instruccions_totals [] = {
98 0, 558309327207, 5421059240140, 186483654001, 0,776504509655,614626187081,
99 480070400876,802635200538,261219407437,428862715907,470328894765,428418848680,
100 886931409787,204234912479,504060702499,463926761740,843934894626,492910117299,
101 498894476043,0,373477511853,628243699177,376876177856,445124040617,0,0,0
102 };
103
104 static void get_counts(perf_event_desc_t *fds, int num) {
105   ssize_t ret;
106   int i;
107
108   //fprintf(stderr, "PMU_VALUES: ");
109
110   for(i=0; i < num; i++) {
111     uint64_t val;
112     ret = read(fds[i].fd, fds[i].values, sizeof(fds[i].values));
113     if (ret < (ssize_t)sizeof(fds[i].values)) {
114       printf("FALLA ALGO.\n");
115       if (ret == -1)
116         err(1, "cannot read values event %s", fds[i].name);
117       else
118         warnx("could not read event%d", i);
119     }
120
121     val = fds[i].values[0];
122
123     //fprintf(stderr, "%20"PRIu64" ", val);
124     pmu_counters[i] += val;
125   }
126   //fprintf(stderr, "\n");
127 }
128
129 int measure(pid_t *pid) {
130   perf_event_desc_t *fds = NULL;
131   int i, j, ret, num_fds = 0;
132   int status;
133
134   ret = perf_setup_list_events(options.events, &fds, &num_fds);
135   if (ret || (num_fds == 0)) {
136     exit(1);
137   }
138
139   fds[0].fd = -1;
140   for(j=0; j < num_fds; j++) {
141     fds[j].hw.disabled = 0; /* start immediately */
142
143     /* request timing information necessary for scaling counts */
144     fds[j].hw.read_format = PERF_FORMAT_SCALE;
145     fds[j].hw.pinned = !j && options.pinned;
146     fds[j].fd = perf_event_open(&fds[j].hw, *pid, -1, (options.group? fds[j].fd
           : -1), 0);
147     if (fds[j].fd == -1) {
148       errx(1, "cannot attach event %s", fds[j].name);
149     }
150   }
```

```
151
152   // Llibera els procesos
153   kill(*pid, 18);
154   waitpid(*pid, &status, WCONTINUED);
155   if (WIFEXITED(status)) {
156     printf("ERROR: command process %d exited too early with status %d\n", *pid,
          WEXITSTATUS(status));
157   }
158
159   usleep(options.delay*1000);
160   //sleep(options.delay);
161
162   // Bloqueja els procesos
163   ret = 0;
164   kill(*pid, 19);
165   waitpid(*pid, &status, WUNTRACED);
166
167   if (WIFEXITED(status)) {
168     printf("Process %d finished with status %d\n", *pid, WEXITSTATUS(status));
169     ret=1;
170   }
171
172   get_counts(fds, num_fds);
173
174   for(i=0; i < num_fds; i++) {
175     close(fds[i].fd);
176   }
177
178   perf_free_fds(fds, num_fds);
179
180   return ret;
181 }
182
183 int inicialitzar() {
184   perf_event_desc_t *fds = NULL;
185   int i, ret, num_fds = 0;
186
187   ret = perf_setup_list_events(options.events, &fds, &num_fds);
188   if (ret || (num_fds == 0)) {
189     exit(1);
190   }
191
192   for (i=0; i < num_fds; i++) {
193     events[i] = strdup(fds[i].name);
194     pmu_counters[i] = 0;
195   }
196
197   for(i=0; i < num_fds; i++) {
198     close(fds[i].fd);
199   }
200
201   perf_free_fds(fds, num_fds);
202   return num_fds;
203 }
204
205 static void usage(void) {
206   printf("usage: task_attach_timeout [-h] [-P] [-g] [-t (temps secs)] [-d delay
        (msecs)] [-e cycles,event2,...] A prgA\n");
207 }
208
209 int main(int argc, char **argv) {
210   int c, ret, i;
211   //int temps = -1;
212   pid_t pid;
```

```
213   int prg;
214   cpu_set_t mask;
215   int status, num_fds;
216   FILE *fitxer;
217   //int max_quantums = -1;
218   int nucli = -1;
219   prg = -1;
220   options.verbose = 0;
221   options.delay = 0;
222
223   while ((c=getopt(argc, argv,"he:d:t:pvgPA:Q:C:")) != -1) {
224     switch(c) {
225       case 'e':
226         options.events = optarg;
227         break;
228       case 'P':
229         options.pinned = 1;
230         break;
231       case 'g':
232         options.group = 1;
233         break;
234       case 'd':
235         options.delay = atoi(optarg);
236         break;
237       case 'h':
238         usage();
239         exit(0);
240       case 'A':
241         prg = atoi(optarg);
242         break;
243       case 't':
244         //temps = atoi(optarg);
245         break;
246       case 'v':
247         options.verbose = 1;
248         break;
249       case 'Q':
250         //max_quantums = atoi(optarg);
251         break;
252       case 'C':
253         nucli = atoi(optarg);
254         break;
255       default:
256         errx(1, "unknown error");
257     }
258   }
259
260   if (!options.events) {
261     options.events = strdup("cycles,instructions,PM_DATA_ALL_FROM_L2,
          PM_DATA_FROM_L2MISS,LLC-PREFETCHES");
262   }
263
264   if (options.delay < 1) {
265     options.delay = 100;
266   }
267
268   if (prg < 0) {
269     fprintf(stderr, "Error: Falta especificar el proces.\n");
270     return -1;
271   }
272
273   if (nucli == -1) {
274     CPU_ZERO(&mask);
275     CPU_SET(48, &mask);
```

```
276     CPU_SET(56, &mask);
277     CPU_SET(64, &mask);
278     CPU_SET(0, &mask);
279   }
280   else {
281     CPU_ZERO(&mask);
282     CPU_SET(nucli, &mask);
283   }
284
285   if (pfm_initialize() != PFM_SUCCESS) {
286     errx(1, "libpfm initialization failed\n");
287   }
288
289   // Preparacio
290   num_fds = inicialitzar();
291
292   do {
293     pid = fork();
294     switch (pid) {
295       case -1: //Error
296         printf("No he pogut crear el fill.\n");
297         return -1;
298
299       case 0: // Fill
300         //Descriptors per als que tenen l'entra per l'entrada estandar
301         switch(prg) {
302           case 4:
303             close(0);
304             fitxer = fopen("/home/jacarvi/working_dir/CPU2006/445.gobmk/data/
                  ref/input/13x13.tst", "r");
305             if (fitxer == NULL) {
306               printf("Error. No s'ha pogut obrir el fitxer 13x13.tst.\n");
307               return -1;
308             }
309             break;
310           case 13:
311             close(0);
312             fitxer = fopen("/home/jacarvi/working_dir/CPU2006/416.gamess/data/
                  ref/input/h2ocu2+.gradient.config", "r");
313             if (fitxer == NULL) {
314               printf("Error. No s'ha pogut obrir el fitxer h2ocu2+.energy.
                    config.\n");
315               return -1;
316             }
317             break;
318           case 14:
319             close(0);
320             fitxer = fopen("/home/jacarvi/working_dir/CPU2006/433.milc/data/ref
                  /input/su3imp.in", "r");
321             if (fitxer == NULL) {
322               printf("Error. No s'ha pogut obrir el fitxer su3imp.in.\n");
323               return -1;
324             }
325             break;
326           case 18:
327             close(0);
328             fitxer = fopen("/home/jacarvi/working_dir/CPU2006/437.leslie3d/data
                  /ref/input/leslie3d.in", "r");
329             if (fitxer == NULL) {
330               printf("Error. No s'ha pogut obrir el fitxer leslie3d.in.\n");
331               return -1;
332             }
333             break;
334           case 22:
```

```
335             close(2);
336             fitxer = fopen("/home/jacarvi/working_dir/povray.sal", "w");
337             if (fitxer == NULL) {
338               printf("Error. No s'ha pogut obrir el fitxer povray.sal\n");
339               return -1;
340             }
341             break;
342         } //Del switch  del case fill
343
344         execv(benchmarks[prg][0], benchmarks[prg]);
345         printf("ERROR EN EL EXEC.\n");
346         return -1;
347
348       default:  //pare
349
350         usleep (200000);  // Esperem 200 ms
351
352         //Asigne el proces a un nucli
353         if (sched_setaffinity(pid, sizeof(mask), &mask) != 0){
354           //en lloc de sizeof ficar 1 anava be
355           printf("Sched_setaffinity error: %d.\n", errno);
356           exit(1);
357         }
358
359         //Parem el proces
360         kill (pid, 19);
361         waitpid(pid, &status, WUNTRACED);
362         if (WIFEXITED(status)) {
363           printf("ERROR: command process %d exited too early with status %d\n",
                   pid, WEXITSTATUS(status));
364         }
365     } //Del switch del execv
366
367     while (pmu_counters[1] < instruccions_totals[prg]) {
368       ret = measure(&pid);
369       if (ret) {
370         fprintf(stderr, "El process ha finalitzat abans de completar tots els
                quantums.\n");
371         break;
372       }
373     }
374   } while (pmu_counters[1] < instruccions_totals[prg]);
375
376   kill(pid, 9);
377
378   fprintf(stderr, "PMU_COUNTS: ");
379
380   for (i=0; i<num_fds; i++) {
381     fprintf(stderr, "%20"PRIu64" ", pmu_counters[i]);
382   }
383
384   fprintf(stderr, "\n");
385
386   /* free libpfm resources cleanly */
387
388   pfm_terminate();
389   return 0;
390 }
```

Listing 2.4: Programa en C

- En la sección de declaración de variables globales (desde la línea 22 a la 102), definimos una serie de estructuras y variables, de las cuales se pueden resumir en:

  - Un struct, *options*, que agrupa los parámetros que se le pasarán en la invocación del programa.
  - Un array de enteros sin signo y de siete posiciones llamado *pmu_counters*, que sirve para almacenar allí los eventos que vamos a medir. Como podemos ver, está limitado a 7 eventos porque solo existen siete contadores hardware y no se soportan más mediciones a la vez. Las instrucciones deben estar siempre en primera posición porque en todo el programa se da por sentado que van a ser éstas las que ocupen tal sitio; es así porque necesitamos hacer referencia en algunos puntos del programa al número de instrucciones, como veremos más adelante.
  - Un array de rutas de benchmarks, para invocar al benchmark que queremos que se ejecute solamente con un parámetro numérico.
  - Un array de cifras unsigned long int, *instruccions_totals* que significa cada una el número de instrucciones que se han ejecutado en 120 segundos. Cada posición del array coincide con las posiciones de los benchmarks, esto es, en dos minutos cada aplicación ha tenido tiempo de ejecutar un número de instrucciones, y así limitaremos en un futuro las ejecuciones a sus números de instrucciones.

- Método *get_counts* (l. 104 a 127). Hace la consulta de los eventos a medir mediante la función *read()* y descriptores de fichero.
- Método *measure* (l.129 a 181). Crea los descriptores de ficheros para medir los eventos, para la ejecución del benchmark y llama al método *get_counts*.
- Método *inicializar* (l.183 a 203). Pone a 0 todo; los descriptores de ficheros, los contadores, etc.
- Método *usage* (l.205 a 207). Da información sobre cómo invocar al programa.
- Método **main** (l.209 a 390). El método principal. Lo explicaremos en secciones:
  - Sección de variables (l.210 a 221).
  - Obtención de parámetros pasados al invocar el programa (l.223 a 258).
  - Si no se han introducido los eventos por línea de comandos, se insertan los eventos que hay por defecto en esa cláusula *if* (l.260 a 262).
  - Si no se ha indicado el delay, se pone 100 ms por defecto (l.264 a 266). Este delay sirve para que el benchmark se detenga el tiempo especificado en delay cuando el programa principal va a recoger los datos en los contadores de prestaciones.
  - Si no se ha indicado un benchmark sobre el cual efectuar la medición, se termina la ejecución del programa (l.268 a 271).
  - Se establece el grupo de procesadores que se van a utilizar (l.273 a 283). En nuestro experimento esta parte del código no se considera, pues solamente ejecutaremos una aplicación a la vez, por lo que solo se considera una parte del procesador.
  - Inicializa el *performance monitoring library*, es decir, el monitorizador de eventos (l.258 a 287).
  - Llama al método *inicialitzar* (l.290).
  - Comienzo de un *do-while*, fork i ejecución del hijo (l.290 a 348). Se hace un fork y el hijo se encarga de lanzar a ejecutar el benchmark. Se mira si es alguno de los casos peculiares, es decir, si necesita algún fichero auxiliar, y se le asigna. En la línea 344 se lanza a ejecutar.

- El padre asigna al hijo a algún procesador del grupo definido. A los 200 ms se para la ejecución (l.348 a 365).
- Bucle (l.367 a 373) cuya condición es que, hasta que no superen el número de instrucciones medidas para el benchmark estudiado o el programa haya finalizado, se medirán los eventos.
- While de la estructura *do-while* (l.374), que sirve para relanzar el benchmark en caso de que haya finalizado antes de haber llegado al número de instrucciones ejecutadas durante 120 segundos en solitario.
- Se mata al proceso hijo y se muestran los resultados finales, fruto de la medición (l.375 a 390).

Contadores de prestaciones

Cuando hablamos de contadores de prestaciones (performance counters) hacemos referencia a registros hardware que capturan aquellos determinados eventos de bajo nivel que ocurren en el procesador y el subsistema de memoria. Algunos ejemplos de eventos son el número de ciclos ejecutados por una aplicación, el número de instrucciones ejecutadas, los fallos en lectura de caché, los aciertos del predictor de saltos, etcétera. Exactamente, los registros en los que se almacenan el número de ocurrencias de los eventos se llaman *Performance Monitor Counters* (PMC), o Contadores del monitor de prestaciones. El procesador sobre el cual estamos realizando el estudio, el IBM Power8, tiene seis registros de PMC por cada núcleo, de los cuales dos son siempre fijos -miden los ciclos y las instrucciones ejecutadas- y los demás son configurables.

Para acceder a estos registros, hemos empleado la herramienta *perf* de Linux, que utiliza los contadores de prestaciones. Esta herramienta la implementa la librería *perf_util.h*, que forma parte del paquete *libpfm*, el cual ayuda a codificar eventos (o prestaciones a medir) para su uso con los sistemas operativos, es decir, para trabajar con las herramientas de perf en Linux. Los eventos que hemos empleado en este estudio los hemos obtenido del proyecto *Oprofile*, el cual ofrece un perfil estadístico para los sistemas Linux. La lista de los eventos que funcionan en Power8 se encuentran en la web de este citado proyecto.

Los contadores de eventos seleccionados para nuestro estudio han sido los de Perf **Cycles** e **Instruccions**, y de Oprofile **PM_DATA_FROM_L3**, **PM_DATA_FROM_L3MISS** y **PM_MEM_PREF**. Estos eventos los situamos siempre en el programa en C, en la línea 261 o los pasamos por parámetro en el script de Shell Script al llamar al programa en C.

Análisis sintáctico con Java

Ejemplos de los resultados obtenidos del lanzamiento del programa en C a través del script se han mostrado en el listing 2.3 de la sección de Shell Script. Estos datos se recogen en una hoja de cálculo para poder operar con ellos, es decir, sumarlos, dividirlos, graficarlos... Para ello, queremos convertirlos a formato CSV para poder abrilos con el programa LibreOffice Calc. Par conseguir esto se ha escrito un programa Java que lee el fichero y lo escribe en otro separando los datos por comas.

```
1 import java.io.*;
2 import java.util.Scanner;
3 class ParserOfResults {
```

```
4   public static void main(String[] args){
5     String[][][] matrix = new String[22][6][6]; //benchmark, unid medidas, dscr
6     int nombre_benchmarks = 22;
7     int nombre_unitats_mesurades = 6; //Siempre sumamos uno porque el 0 no
          cuenta
8     int nombre_dscr = 6;
9
10    for(int i = 0; i < 5; i++){matrix[0][1][i] = "Cicles";}
11    for(int i = 0; i < 5; i++){matrix[0][2][i] = "Instruccions";}
12    for(int i = 0; i < 5; i++){matrix[0][3][i] = "LLC-LOADS";}
13    for(int i = 0; i < 5; i++){matrix[0][4][i] = "LLC-LOAD-MISSES";}
14    for(int i = 0; i < 5; i++){matrix[0][5][i] = "PM_MEM_PREF";}
15
16    Scanner scan = null;
17    try{
18      scan = new Scanner (new File("Memoria.dat"));
19      int index_matrix = 1; //0 no porque es la primera fila de nombres
20      int quin_dscr = -1; //cual de todas las matrices
21      String num_benchmark = ""; //numero del benchmark
22
23      while(scan.hasNextLine()){
24
25        String liniaactual = scan.nextLine();
26
27        /*Linea de # benchmark = dscr*/
28        if(liniaactual.charAt(0) == '#'){
29          String[] valors = liniaactual.substring(1).trim().split("=");
30          valors[0] = valors[0].trim();
31          valors[1] = valors[1].trim();
32          num_benchmark = valors[0];
33          quin_dscr = transformaIndexDSCR(Integer.parseInt(valors[1]));
34
35          if (valors[0].equals("1")) {
36            index_matrix = 1;
37          } else {
38            index_matrix++;
39          }
40
41        /*Linea de PMU_COUNTS: ...*/
42        }
43        if(liniaactual.length() > 11){
44          if(liniaactual.startsWith("PMU_COUNTS")){
45            String cadtemp = liniaactual.substring(11).trim();
46            String[] arrayValors = cadtemp.split(" ");
47            matrix[index_matrix][0][quin_dscr] = num_benchmark; //Asignamos el
                  numero de benchmark a la fila
48            //Ahora hacemos correr, en la pagina del dscr, en la fila que
                  pertoca, los valores en las columnas
49            int j = 1;
50            for(int i = 0; i < arrayValors.length; i++){
51              if (!arrayValors[i].isEmpty() & !arrayValors[i].equals(" ")){
52                matrix[index_matrix][j][quin_dscr] = arrayValors[i];
53                j++;
54              }
55            }
56          }
57        }
58      }
59
60      PrintWriter pw = null;
61      for (int i = 0; i < nombre_dscr; i++){
62        pw = new PrintWriter("resultado"+i+".csv");
63        for (int j = 0; j < nombre_benchmarks; j++){
64          for (int k = 0; k < nombre_unitats_mesurades; k++){
```

```
65            /* Si ultima linea */
66              if (k == nombre_unitats_mesurades-1) {
67                pw.println("\""+matrix[j][k][i]+"\"");
68              } else {
69                pw.print("\""+matrix[j][k][i]+"\",");
70              }
71            }
72          }
73          pw.close();
74        }
75      } catch (Exception e){
76        e.printStackTrace();
77      } finally {
78        scan.close();
79      }
80    }
81
82    private static int transformaIndexDSCR(int a){
83      switch(a){
84        case 0: return 0;
85        case 66: return 1;
86        case 455: return 2;
87        case 71: return 3;
88        case 450: return 4;
89        case 1: return 5;
90      }
91      return -1;
92    }
93  }
```

Listing 2.5: Programa en Java de conversión a CSV

Como se aprecia, primero se almacenan en arrays las prestaciones medidas y después se imprimen recorriendo las matrices. Elaboramos este programa de esta forma porque en caso de querer cambiar la manera de mostrar los datos, era tan fácil como empezar a recorrer el array desde otros puntos de partida.

CAPÍTULO 3
Desarrollo del estudio

Normalización de las medidas

Cuando hablamos de las medidas tomadas nos referimos a la obtención de los contadores de eventos, los cuales cuentan las veces en que se ha dado el evento sobre la ejecución de uno de los benchmarks con una determinada configuración del prefetcher. Si estas medidas se tomaran sobre un determinado benchmark y se compararan respecto a otro benchmark, no sería correcto del todo, pues cada aplicación tiene un tiempo de ejecución diferente. Por este motivo, se utilizó una metodología de amplio uso en los estudios científicos. Las pruebas duran el mismo tiempo (en nuestro caso dos minutos cada una), haciendo que las que duren menos se relancen, y las que duren más finalicen su ejecución. Asimismo, debe resolverse también la siguiente cuestión. Si un conjunto de configuraciones provoca que una aplicación tenga mejor o peor rendimiento —lo cual ralentiza o acelera la ejecución—, esto afectará a la temporización de las ejecuciones. Entonces, no se pueden limitar las ejecuciones de los benchmarks a únicamente una medida temporal, pues esto nos cortaría las pruebas.

Para resolver esta situación hemos decidido ejecutar las pruebas dos minutos cada benchmark en solitario. En este experimento se han tomado como medidas los ciclos y las instrucciones ejecutadas, todo con una configuración del prefetcher por defecto, es decir, la 0. Con esta prueba se obtienen el número de instrucciones ejecutadas por cada aplicación en dos minutos, y en cada nueva ejecución del benchmark para medir los otros contadores se fuerza que el número de instrucciones ejecutado no supere el número de instrucciones ejecutadas durante dos minutos en solitario. El momento en que se supera esta cifra, se detiene la ejecución del benchmark. Las instrucciones medidas se encuentran en el array del programa en C (listing 2.4, l.97 hasta 102) y sus posiciones en la matriz corresponden al número de benchmark en la matriz de los benchmarks (listing 2.4, l.38 hasta 95)

Rendimiento

Para saber si una aplicación tiene buen rendimiento o no, se ha empleado como métrica de prestaciones el **IPC** o **Instrucciones por Ciclo**. Esta medida se obtiene de la siguiente forma:

$$IPC = \frac{\#\ Instrucciones\ ejecutadas}{\#\ Ciclos\ de\ ejecucion} \tag{3.1}$$

Con esta medida se sabe que, a más IPC, mejor rendimiento tiene una aplicación, pues más instrucciones se han ejecutado de media por ciclo; además, en nuestro caso los ciclos coinciden para las distintas aplicaciones.

Los contadores de eventos empleados en este experimento se han elegido del paquete *Perf*, siendo *cycles* e *instructions*, que significan ciclos transcurridos durante la ejecución e instrucciones ejecutadas. Recordemos que se han ejecutado todas las aplicaciones limitando a que no se supere el número de instrucciones que se ejecutan con la configuración por defecto durante dos minutos.

En las tablas 3.1 y 3.2 se presentan unos gráficos de barras con los IPC que han resultado de la ejecución de los benchmarks con diferentes configuraciones del prefetcher, para los benchmarks enteros y de coma flotante, respectivamente. Se presentan en gráficos distintos debido a que no emplean los mismos recursos en el procesador y tienen comportamientos diferentes.

La elección de estas configuraciones del prefetch es debido a que algunos campos del DSCR tienen más impacto en la ejecución que otros. Para nuestro estudio hemos escogido los campos DPFD, que es la profundidad, y URG, que hace referencia a la urgencia. En la tabla 3.1 se muestra en la columna DSCR la configuración en decimal. Dado que la máquina es *little endian*, tenemos que codificar al revés los bits que hay en la figura 1.1, tal y como se cuenta que se haga en este foro de IBM [foro]. En las gráficas de barras se evaluan distintas configuraciones cuyo significado viene explicado en la tabla 3.1.

DSCR	URG	DPFD	Descripción
0	0, por defecto	0, por defecto	Prefetch por defecto
66	1, desactivada	2, profundidad mínima	El prefetch menos agresivo
455	7, el más urgente	7, el más profundo	El prefetch más agresivo
71	1, desactivada	7, el más profundo	Sin urgencia, el prefetch más profundo
450	7, el más urgente	2, profundidad mínima	El más urgente, con la mínima profundidad
1	0, por defecto	1, desactivado	Al desactivar la profundidad, se desactiva todo el prefetch

Tabla 3.1: Explicación de las configuraciones del DSCR

Como podemos observar en la figura 3.1, la ausencia de prefetch está marcada con una barra roja a la derecha del conjunto, siendo el DSCR = 1. Es importante fijarse en ésta y compararla con el resto, porque hay veces donde el rendimiento se aprecia que es significativamente mejor cuando hay prefetch, por ejemplo, en el benchmark Libquantum. Hay benchmarks en los que la diferencia es solo un poco mejor, como Gcc, Mcf, Hmmer; benchmarks en los que no parece afectar el prefetch, siendo estos Bzip2, Gobmk, Sjeng, H264Ref y Astar, y benchmarks que, por algún motivo, el prefetch les empeora la ejecución, como Omnetpp y Xalancbmk. Más adelante se razona el porqué de este comportamiento, anómalo en el último caso.

En la figura 3.2 podemos ver que se estructura del mismo modo que en la gráfica de los benchmarks enteros. Comparando la barra de ausencia del prefetch con las demás, que tiene prefetch, encontramos que hay tres grupos de aplicaciones atendiendo al com-

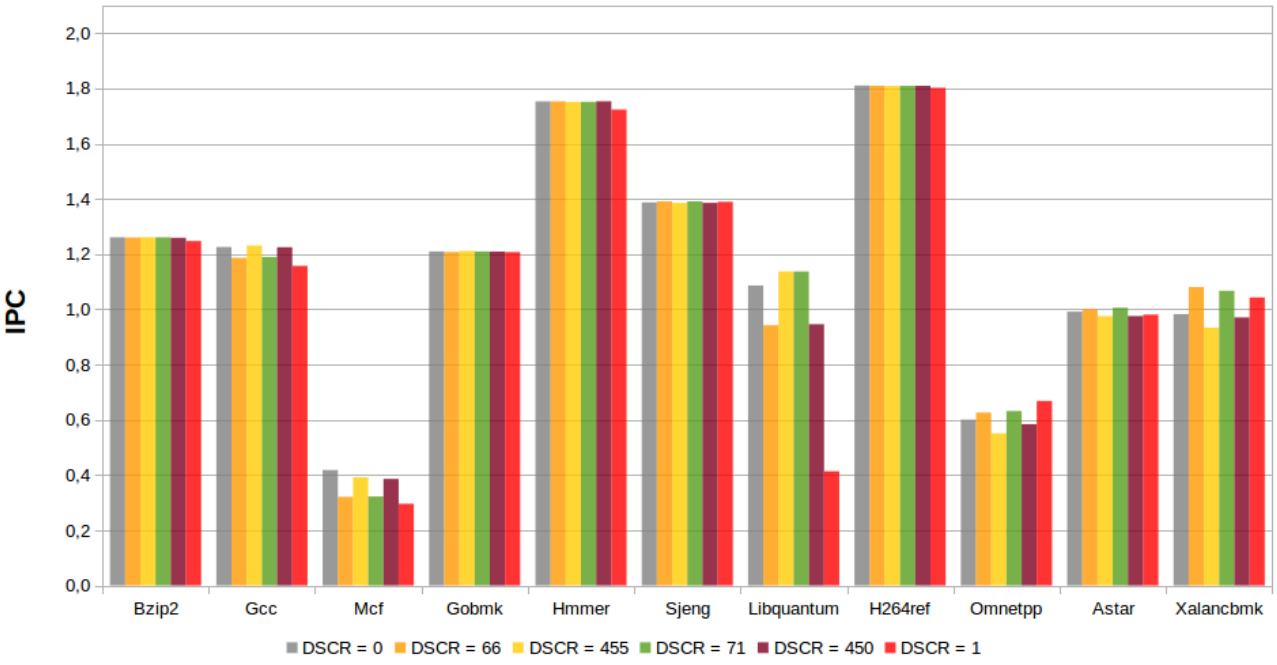

Figura 3.1: IPC de los benchmarks enteros para diferentes configuraciones del DSCR.

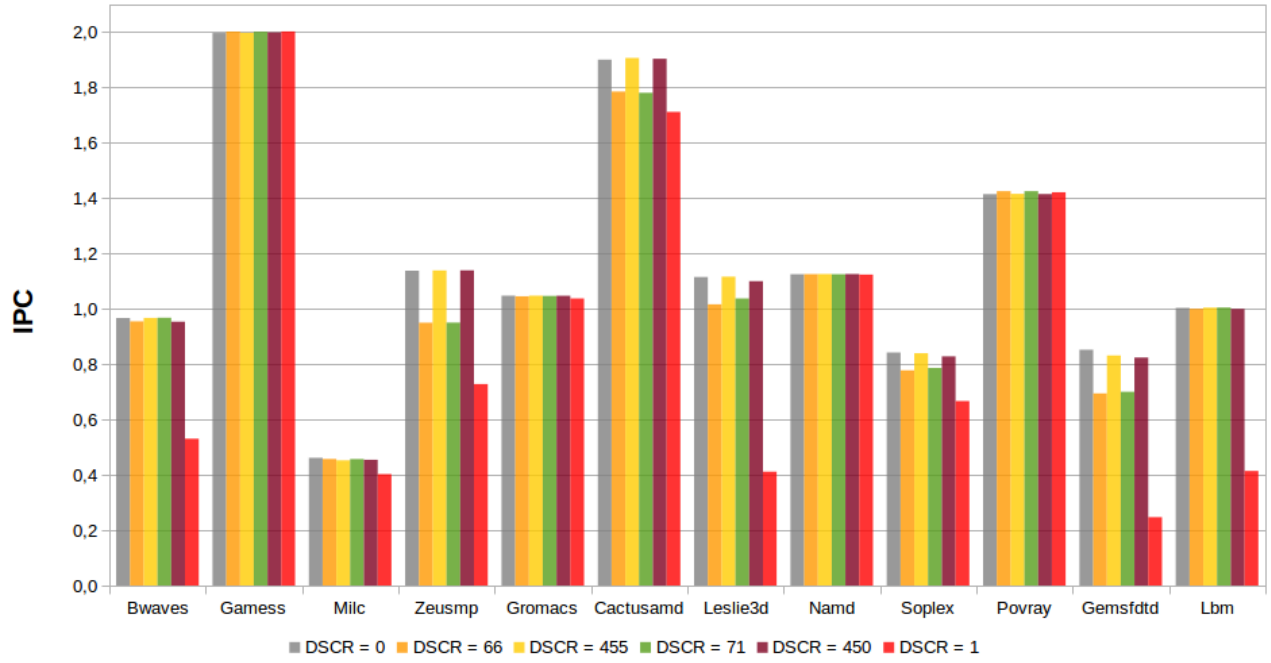

Figura 3.2: IPC de benchmarks de coma flotante para diferentes configuraciones del DSCR.

portamiento que presentan: Aquellas aplicaciones en las que el prefetch beneficia significativamente sus prestaciones, como Bwaves, ZeusMP, Cactusamd, Leslie3d, Soplex, Gemsfdtd y Lbm; los benchmarks a los que el prefetch les afecta positivamente pero de manera moderada, como Milc, y las aplicaciones a las que el prefetch no parece afectarles, como Gamess, Gromacs, Namd, y Povray. También justificaremos este comportamiento más adelante.

Para apreciar mejor la mejora respecto a la ausencia del prefetch, se presentan unas gráficas con el *speedup*, o aceleración, de las diferentes configuraciones normalizadas respecto al prefetch desactivado. Para las aplicaciones de enteros, los resultados se muestran en la Figura 3.3 y para los speedups de coma flotante, en la Figura 3.4.

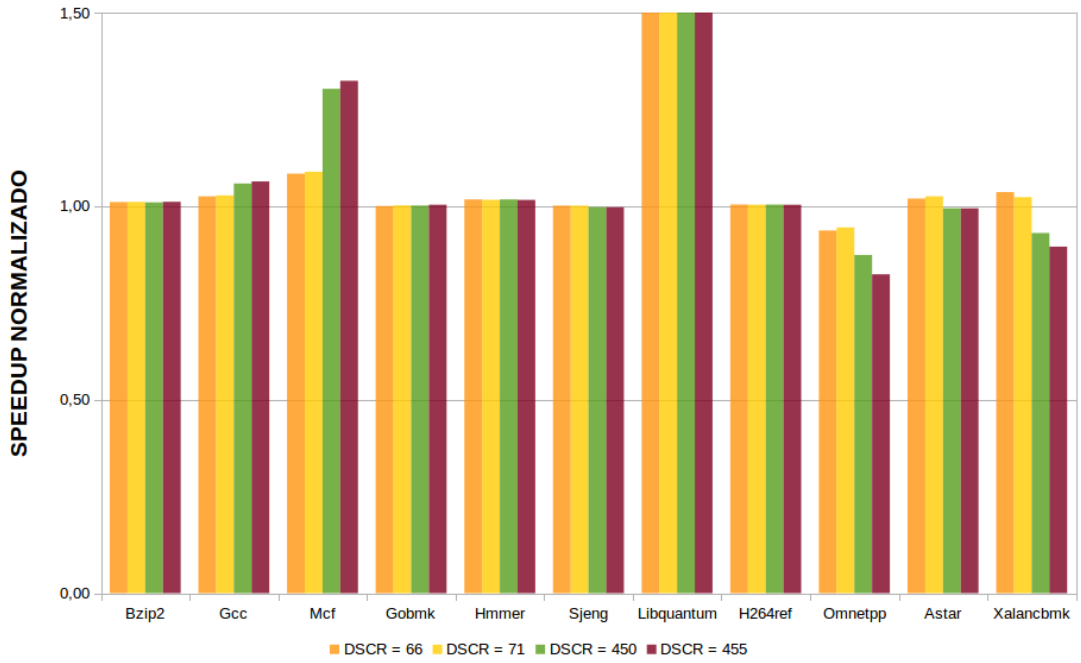

Figura 3.3: Speedup de benchmarks enteros para diferentes configuraciones del DSCR respecto al prefetch desactivado.

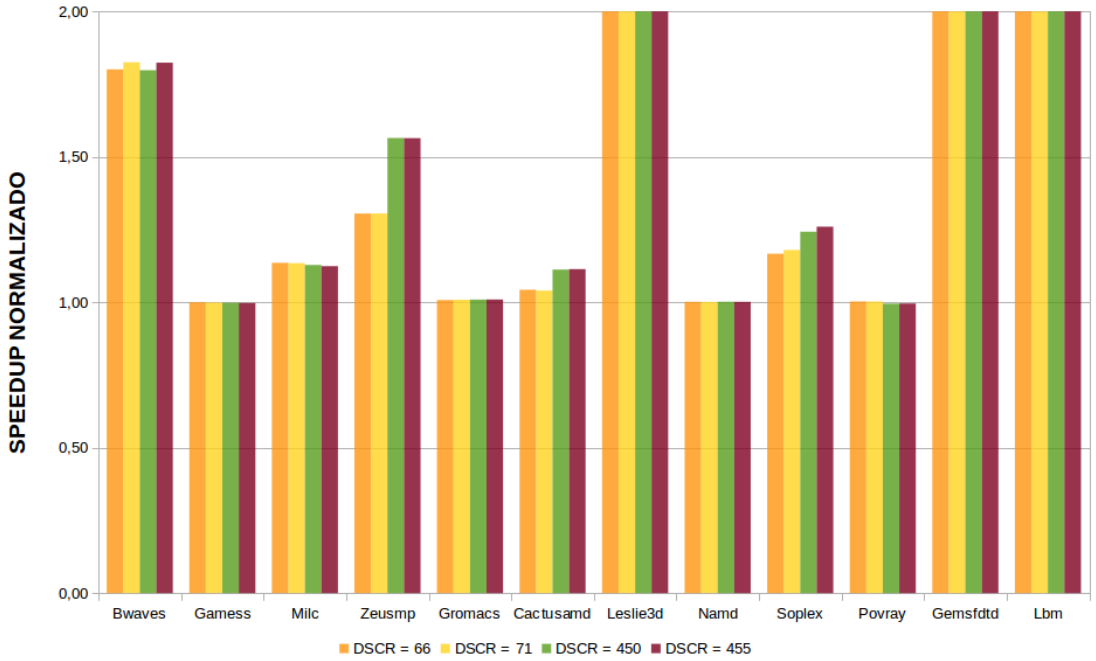

Figura 3.4: Speedup de benchmarks de coma flotante para diferentes configuraciones del DSCR respecto al prefetch desactivado.

Justificación del rendimiento en aplicaciones de enteros

Una de las principales razones por la cuales un sistema de procesamiento puede ir lento es las latencias en el acceso a los datos en la jerarquía de memoria. Las latencias se miden en ciclos y, teniendo en cuenta que nuestra máquina funciona a una frecuencia de 2.06 GHz, un ciclo supone $4{,}8543689^{-10}$ segundos = 0.485 ns. En el IBM Power8, la latencia para L1 es de entre 3 y 5 ciclos (1.455 ns - 2,425 ns), dependiendo de la complejidad; para L2, de 12 ciclos (5,82 ns); para L3, o LLC (*Lowest Level Cache*) son de 27 ciclos (13,095 ns) y para memoria principal, son 80 ns.

La memoria principal es la más lenta, por lo que los accesos que se realizan a ella son los que más penalizan al rendimiento. Pero por otro lado, los accesos a la caché L3 son más abundantes y afectan también significativamente a las prestaciones del sistema. Los fallos son aquellos accesos que se hacen en el sistema de memoria debido a que no se encuentra el dato solicitado en el nivel superior de la jerarquía de memoria. Así, si estudiamos el MPKI (*Misses per Kilo Instruction*, o Fallos por cada mil instrucciones ejecutadas) en la caché L3, se puede apreciar el efecto producido. Se utiliza el MPKI en vez de simplemente el MPI (fallos por instrucción) debido a que el número de fallos es un valor muy pequeño. En la figura 3.5 se muestra el efecto de esta penalización.

$$MPKI_{L3} = 1000 \cdot \frac{Fallos\ L3}{Instrucciones\ ejecutadas} \tag{3.2}$$

En la gráfica 3.5 del MPKI se cortan los datos de mcf; esto está hecho para facilitar la apreciación del resto de datos ya que los datos de mcf son del orden de 20 MPKI. A continuación se razona con esta gráfica el rendimiento de las aplicaciones:

- **Libquantum**: Esta aplicación experimenta un alto speedup, como se observa en la figura 3.3. Se aprecia en el valor de MPKI que la ejecución sin prefetch provoca una cantidad muy alta de fallos, y es por eso que las configuraciones con prefetch de este benchmark tienen un alto IPC comparado con la configuración sin prefetch.

- **Gcc, Mcf**: El speedup que experimentan estas aplicaciones supera por poco al 1, que significa que hay speedup, pero no demasiado, porque si nos fijamos en su MPKI cuando no hay prefetch, es ligeramente superior al resto de configuraciones del benchmark que sí que tienen el prefetch activado. En Mcf la configuración sin prefetch llega a 22,59, y el resto oscilan entre 10 y 20 MPKI.

- **Bzip2, Hmmer, Gobmk, Sjeng, H264Ref, Astar**: Apreciamos que su speedup está en 1 o muy cerca, es decir, no hay speedup, pero por contra, algunos MPKI no guardan relación con el speedup obtenido; deberían ser los mismos MPKI para todas las configuraciones, pero podemos ver, por ejemplo, que no sucede para Astar, como el más destacado.

- **Omnetpp, Xalancbmk**: El speedup sale inferior a 1, es decir, tienen un peor rendimiento con el prefetch activado que sin él. Sin embargo, las configuraciones sin prefetch resultan con un MPKI más elevado todavía que las demás, cosa que no tiene sentido, pero lo veremos más adelante.

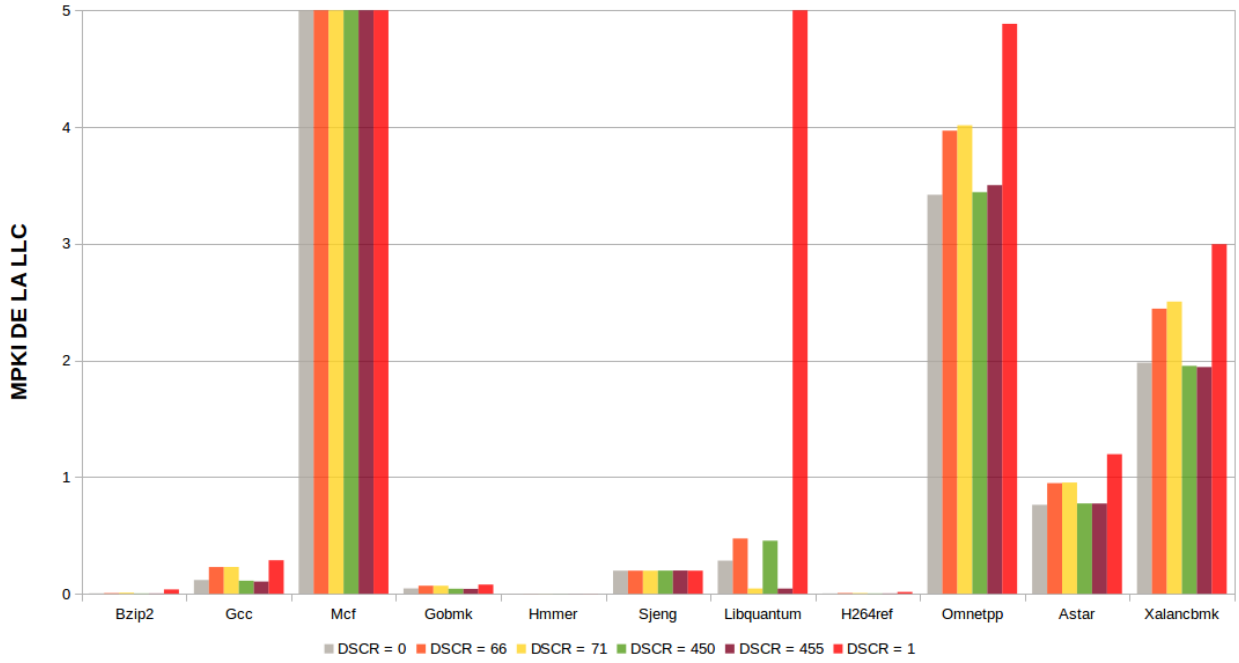

Figura 3.5: MPKI de LLC de benchmarks enteros para diferentes configuraciones del DSCR.

Ahora vemos que los fallos en L3 no son indicativos, ya que ocurren o bien porque el prefetch no predice bien y falla al buscar el dato en la caché, o bien por otros motivos. Vamos a ver el número de lecturas que se hacen, es decir, el número de búsquedas de un dato en las cachés o en la memoria. Hemos medido los accesos tanto a la caché L3, en la figura 3.6, como las realizadas a memoria principal cuyos valores se muestran en la figura 3.7.

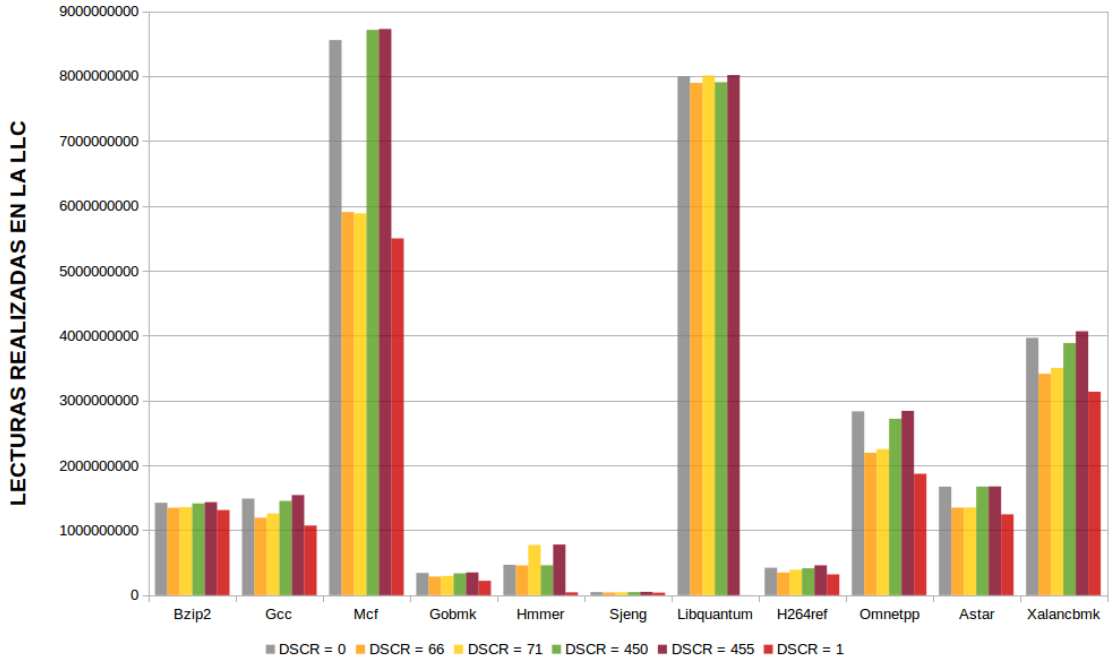

Figura 3.6: Lecturas realizadas en LLC de benchmarks integers para diferentes configuraciones del DSCR.

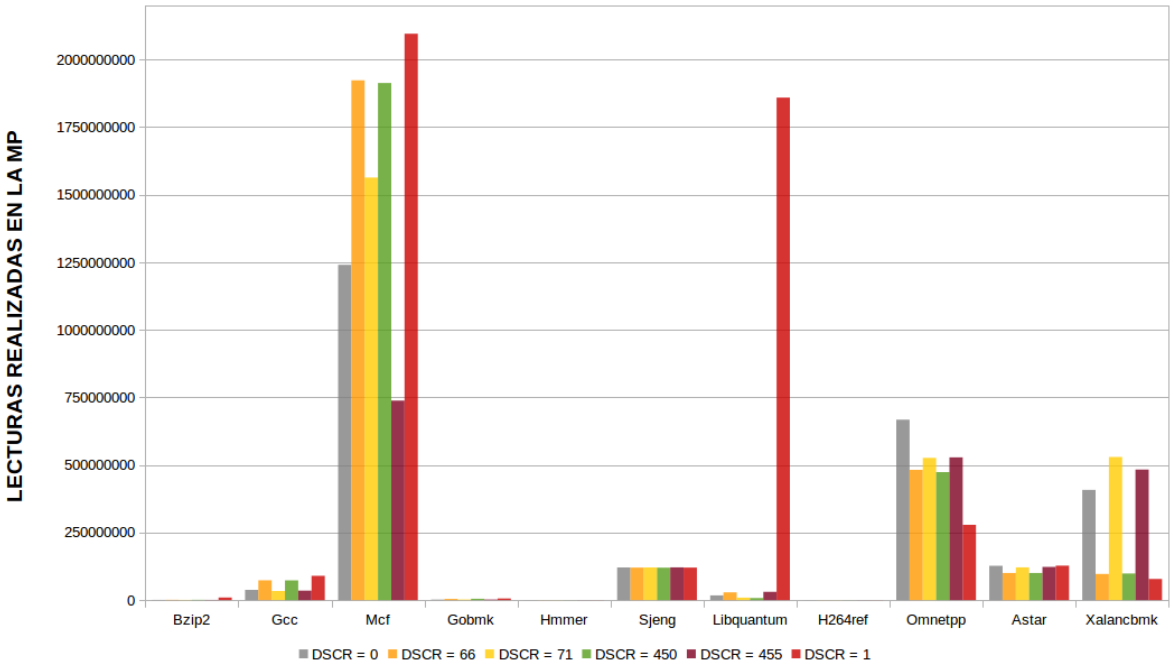

Figura 3.7: Lecturas realizadas en MP de benchmarks integers para diferentes configuraciones del DSCR.

Para ello, hemos utilizado los contadores de eventos **PM_DATA_FROM_MEM** y **PM_DATA_FROM_L3**. En ellos solo se consideran las lecturas regulares pero no se incluyen las lecturas de prefetch en ellas. Para averiguarlo, tuvimos que hacer un pequeño experimento que se detalla en el capítulo 4, relativo al Microbenchmark.

Lo que se observa en la dos figuras de las lecturas es que la L3 tiene muchísimos más accesos de lectura que la memoria principal. Hay que fijarse en la escala para ello: el

límite superior en la gráfica de la L3 es 9.000.000.000 mientras que en la de memoria principal es 2.000.000.000. Por contra, las lecturas de memoria principal pesan mucho más temporalmente que las lecturas de L3, siendo $\frac{80\ ns}{13{,}095\ ns} = 6{,}1$ veces más lenta la memoria principal.

Esto se nota en que, en la gráfica de L3 no parece haber ninguna relación con los IPCs y sus speedups; siguen habiendo menos lecturas con el prefetch desactivado que con el prefetch activado en todos los benchmarks. Por otro lado, en la gráfica de memoria principal se aprecia algo bastante interesante: las aplicaciones que tienen un IPC mayor con el prefetch desactivado que con el prefetch activado (Omnetpp y Xalancbmk) tienen muchas menos lecturas sin prefetch activado que con él activado. Esto nos sugiere que el prefetch (y sus interferencias) **les perjudica**, ya que no se precargan los bloques que serían convenientes. En cambio, en Libquantum, que es la aplicación que más beneficio obtiene del prefetch, se observa que sin prefetch es el que más pérdidas tiene, y con prefetch apenas tiene. Esta es una aplicación a la que sí que **beneficia** el prefetch.

Queda justificado que, la **cantidad total de lecturas en memoria principal** que tienen que realizar las aplicaciones es lo que realmente define el comportamiento en general de los rendimientos de las aplicaciones de enteros.

Comparación entre configuraciones del DSCR

A continuación vamos a mostrar una tabla con el IPC exacto que se obtienen con las diferentes configuraciones del prefetcher.

	DSCR = 0	DSCR = 66	DSCR = 71	DSCR = 450	DSCR = 455
Bzip2	1,26	1,26	1,26	1,26	1,26
Gcc	1,23	1,19	1,19	1,22	1,23
Mcf	0,42	0,32	0,32	0,39	0,39
Gobmk	1,21	1,21	1,21	1,21	1,21
Hmmer	1,75	1,75	1,75	1,75	1,75
Sjeng	1,39	1,39	1,39	1,39	1,39
Libquantum	1,09	0,94	1,14	0,95	1,14
H264ref	1,81	1,81	1,81	1,81	1,81
Omnetpp	0,60	0,63	0,63	0,58	0,55
Astar	0,99	1,00	1,00	0,98	0,98
Xalancbmk	0,98	1,08	1,07	0,97	0,93

Tabla 3.2: IPC en benchmarks integers en las configuraciones del DSCR

A continuación, se consideran los grupos que hemos hecho antes sobre el speedup del prefetch, y relacionaremos las mejores configuraciones con los grupos:

- **Libquantum**: Se observa que la mejor configuración es aquella que tenga la mayor profundidad. Por ello, deducimos que el prefetch le beneficia y mucho; cuanto más se recorra con el prefetch, mejor.
- **Gcc, Mcf**: Un prefetch medio es lo mejor para estos benchmarks. Obtienen un speedup pequeño, pero lo obtienen. Así que es lo mejor tener tanto una urgencia media como una profundidad media.

- **Bzip2, Hmmer, Gobmk, Sjeng, H264Ref, Astar**: No les afecta apenas el prefetcher; como no tienen speedup, tampoco les afecta la ausencia de éste.
- **Omnetpp, Xalancbmk**: El prefetch les perjudica, siendo que la ausencia de prebúsqueda les beneficia. Dentro de las variedades de prefetch, la más ligera es la que mejor les viene, cosa lógica si es que les perjudica.

Cobertura y precisión

De todas las piezas del subsistema de memoria, hemos decidido evaluar el rendimiento a través de la memoria principal porque es su latencia la que más pesa sobre las prestaciones finales. Para ello, tomaremos dos métricas para juzgar los accesos a memoria y las relacionaremos con el rendimiento: **Accuracy** y **Coverage**. El accuracy, o precisión, es la fracción del prefetch que ha resultado útil. La fórmula teórica de este cálculo se obtenido, considerando los contadores existentes, como sigue:

$$Accuracy = \frac{\#\ Aciertos\ de\ prefetch}{\#\ Total\ de\ prefetches} \tag{3.3}$$

Expresándolo con unidades que podamos medir con contadores de eventos:

$$Accuracy = \frac{Fallos\ MP\ eliminados\ por\ prefetch}{(Total\ Prefetch - Fallos\ MP\ elim.\ por\ pref.) + (Fallos\ MP\ elim.\ por\ pref.)} \tag{3.4}$$

$$\textbf{Accuracy} = \frac{\textbf{Accesos MP Sin Prefetch} - \textbf{Accesos MP Con Prefetch}}{\textbf{Total Prefetch}} \tag{3.5}$$

A continuación se analiza la fórmula del coverage, o cobertura en castellano. La cobertura es la fracción de fallos de caché eliminados gracias a la prebúsqueda. Éste se mide con esta ecuación:

$$Coverage = \frac{Accesos\ MP\ eliminados\ por\ prefetch}{Accesos\ MP\ totales} \tag{3.6}$$

Y aplicada con unidades medibles por contadores de eventos, sería:

$$\textbf{Coverage} = \frac{\textbf{Accesos MP Sin Prefetch} - \textbf{Accesos MP Con Prefetch}}{\textbf{Accesos MP Sin Prefetch}} \tag{3.7}$$

Para ello, hemos utilizado PM_DATA_FROM_L3MISS, que son los fallos de peticiones de lectura en la caché L3, que también significa que son los accesos de realizados a la Memoria Principal debido a que no se ha encontrado el dato en la LLC. También hemos utilizado PM_MEM_PREF, que son los accesos a memoria principal para prefetch.

Podemos deducir de los resultados de la precisión presentados en la tabla 3.3 que la **precisión es mejor generalmente cuando está configurado el DSCR para estar al mínimo de urgencia**, es decir, DSCR = 66 y DSCR = 71. Esto es normal que se muestre así porque el número de prefetches generados cuando las configuraciones están al máximo de la urgencia - el 450 y 455 - es demasiado alto, y el ahorro en lecturas a MP se reduce,

	DSCR = 0	DSCR = 66	DSCR = 71	DSCR = 450	DSCR = 455
Bzip2	0,78	1,02	0,79	0,87	0,82
Gcc	0,15	0,27	0,26	0,13	0,13
Mcf	0,23	0,78	0,75	0,18	0,17
Gobmk	0,46	0,70	0,69	0,43	0,40
Hmmer	0,15	0,15	0,51	0,14	0,07
Sjeng	0,00	0,02	-0,02	0,00	0,00
Libquantum	2,07	1,88	1,82	2,00	1,86
H264ref	1,27	3,92	1,64	0,96	0,59
Omnetpp	0,03	0,14	0,14	0,03	0,02
Astar	0,08	0,70	0,68	0,06	0,05
Xalancbmk	0,06	0,34	0,30	0,06	0,04

Tabla 3.3: Precisión del prefetcher en aplicaciones de números enteros en las configuraciones del DSCR

	DSCR = 0	DSCR = 66	DSCR = 71	DSCR = 450	DSCR = 455
Bzip2	0,93	0,87	0,76	0,93	0,94
Gcc	0,59	0,20	0,20	0,61	0,64
Mcf	0,57	0,09	0,09	0,61	0,61
Gobmk	0,42	0,13	0,12	0,46	0,47
Hmmer	0,23	0,03	0,09	0,18	0,30
Sjeng	0,00	0,00	0,00	0,00	0,00
Libquantum	0,98	0,97	1,00	0,97	1,00
H264ref	0,87	0,64	0,63	0,86	0,86
Omnetpp	0,30	0,19	0,18	0,30	0,28
Astar	0,36	0,21	0,20	0,35	0,35
Xalancbmk	0,34	0,18	0,16	0,35	0,35

Tabla 3.4: Cobertura del prefetcher en aplicaciones de números enteros en las configuraciones del DSCR

pues el número de prefetches está en el denominador de la ecuación; provoca que a mayor número de prefetches, menor precisión.

Por otro lado, en la tabla 3.4 de la cobertura vemos que la **cobertura es habitualmente más alta cuando aumentan la urgencia, y algunas veces la profundidad**; Era de esperar, pues más accesos a memoria se hacen en prefetch y no accesos regulares, lo que reduce la cantidad de accesos en lecturas normales en memoria principal provocados por fallos en la caché L3.

Justificación del rendimiento en aplicaciones de coma flotante

Una diferencia sustancial entre las aplicaciones de enteros y las de coma flotante es que las aplicaciones, por lo general, tienden a aprovechar más el prefetch. Hay muchas aplicaciones que sacan beneficio del prefetch, es decir, hay una diferencia notable entre las prestaciones de las aplicaciones ejecutadas con prefetch y las prestaciones obtenidas con el prefetch desactivado. Esto lo hemos visto en la gráfica del speedup para benchmarks de coma flotante en la figura 3.4. De manera análoga al análisis de aplicaciones de enteros, los grupos son:

- **Bwaves, ZeusMP, Milc, Leslie3D, Soplex, GemsFDTD** y **Lbm**. Todas estas aplicaciones experimentan un notable speedup. De esto hablábamos cuando decíamos anteriormente que había muchas aplicaciones que experimentaban una alta aceleración respecto a no tener prebúsqueda.
- **CactusAMD**. Tiene speedup mayor que uno, pero tampoco es muy elevado.
- **Gamess, Gromacs, Namd** y **Povray**. El prefetch no les afecta en nada, les deja indiferentes.

Como hemos visto en el primer grupo, abundan los benchmarks a los que les ocurre. La alta efectividad del prefetch en los benchmarks de coma flotante es debido a que tratan grandes matrices de datos, con lo que exhiben un alto paralelismo a nivel de bucle. El patrón suele ser regular y fácil de detectar por el prefetcher. Llamamos a este fenómeno *Loop Level Parallelism*, o paralelismo a nivel de bucle.

A continuación, se analiza el MPKI, como antes hemos hecho. La gráfica que muestra los resultados del estudio es la figura 3.8. En este caso, no tenemos aplicaciones a las que el prefetch les empeore la ejecución, por lo cual, no podemos ver claramente lo que pasaba con los enteros: el MPKI no resultaba esclarecedor para saber qué empeora el rendimiento del prefetch. Además, las aplicaciones a las que les resulta indiferente el prefetch, no tienen apenas MPKI, por lo que no se puede discernir si tiene más o menos MPKI con una configuración u otra.

Finalmente, de manera análoga a la sección de las aplicaciones de números enteros, se analizan las lecturas, en la figura 3.9. Como ha ocurrido con el MPKI, las aplicaciones a las cuales el prefetch no afecta (Gamess, Gromacs, Namd y Povray) no tienen apenas lecturas en la memoria. Esto es lógico porque si no hay apenas fallos en L3, menos accesos va a haber incluso en MP. De todos modos, aunque se vea con el MPKI de la LLC en esta sección, debemos tener en cuenta a las aplicaciones de enteros; también se ve la justificación con las lecturas en MP en este caso, así que definitivamente, las **lecturas a memoria principal son la causa de un rendimiento mayor o peor en las diferentes configuraciones del prefetch**.

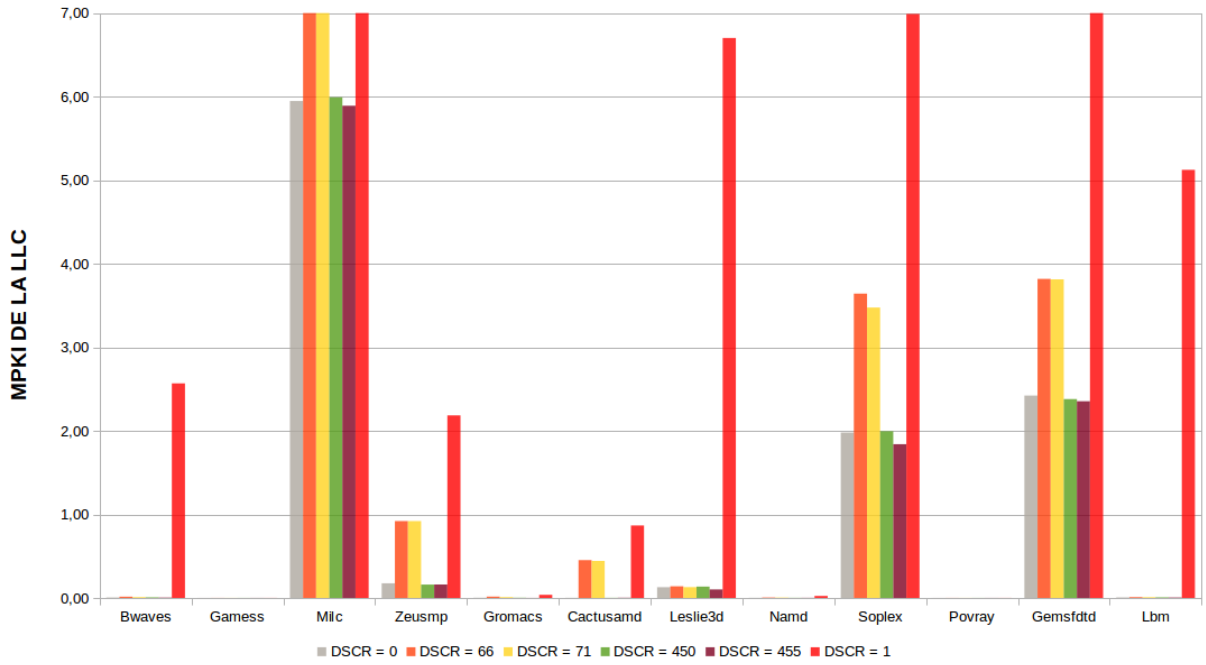

Figura 3.8: MPKI de LLC de benchmarks de coma flotante para diferentes configuraciones del DSCR

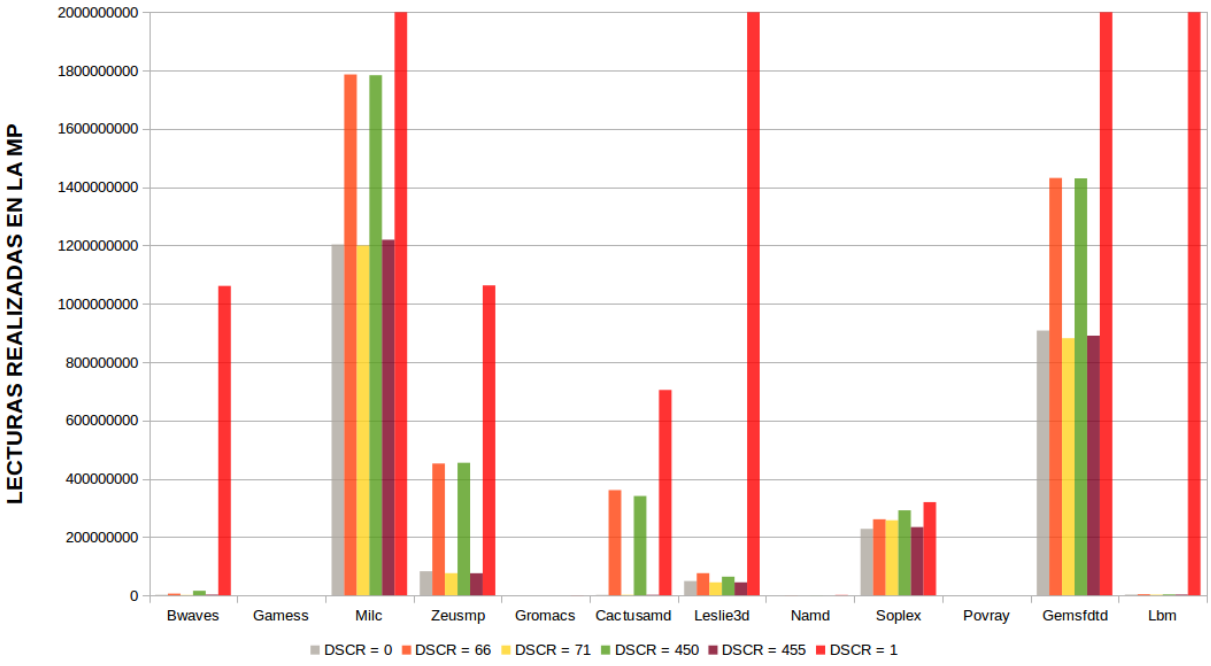

Figura 3.9: Accesos de lectura a MP de benchmarks de coma flotante para diferentes configuraciones del DSCR

Comparación entre configuraciones del DSCR

A continuación se muestra en la tabla 3.5 el IPC exacto que se obtiene con las diferentes configuraciones del prefetcher. Resaltaremos en azul los índices de IPC más altos según el benchmark, con el objetivo de resaltar qué beneficia más a cada aplicación. Aquellas aplicaciones que no tienen ningún resalte, es debido a que su IPC es igual o casi en todas las configuraciones.

	DSCR = 0	DSCR = 66	DSCR = 71	DSCR = 450	DSCR = 455
Bwaves	0,97	0,96	0,97	0,95	0,97
Gamess	2,00	2,00	2,00	2,00	2,00
Milc	0,46	0,46	0,46	0,45	0,45
Zeusmp	1,14	0,95	0,95	1,14	1,14
Gromacs	1,05	1,05	1,05	1,05	1,05
Cactusamd	1,90	1,79	1,78	1,90	1,91
Leslie3d	1,12	1,02	1,04	1,10	1,12
Namd	1,13	1,13	1,13	1,13	1,13
Soplex	0,84	0,78	0,79	0,83	0,84
Povray	1,41	1,43	1,43	1,42	1,42
Gemsfdtd	0,85	0,69	0,70	0,82	0,83
Lbm	1,00	1,00	1,00	1,00	1,00

Tabla 3.5: IPC en benchmarks de coma flotante en las configuraciones del DSCR

Ahora, tomamos los grupos que hemos realizado antes sobre el speedup del prefetch, y se relacion las mejores configuraciones con los grupos:

- **Bwaves, ZeusMP, Milc, Leslie3D, Soplex, GemsFDTD** y **Lbm**. Algunos obtienen mejoras con un prefetch más agresivo, como ZeusMP, y otros no, como Bwaves. Esto no indica nada, ni siquiera sigue un orden en la magnitud del speedup, porque los que más speedup alcanzan (GemsFDTD y Lbm) tienen, uno más IPC en más agresividad, y el otro es igual en todas las configuraciones.
- **CactusAMD**. Parece que un prefetch más agresivo le viene mejor.
- **Gamess, Gromacs, Namd** y **Povray**. El prefetch no les afecta, y tampoco la ausencia de éste. Como ya se ha mencionado anteriormente, este grupo es indiferente ante la presencia de prebúsqueda.

Cobertura y precisión

Como hemos hecho en la sección de la cobertura y precisión en enteros, los hemos estimado según las fórmulas presentadas, con los eventos PM_DATA_FROM_L3MISS, para medir los accesos a memoria principal de lectura regular, y PM_MEM_PREF, para medir los accesos a memoria principal como prebúsqueda. Para los benchmarks de coma flotante hemos plasmado el resultado en las tablas de la precisión 3.6 y la cobertura 3.7.

De manera análoga a las aplicaciones de enteros, las mejores precisiones del prefetcher se encuentran entre las configuraciones de prefetch menos agresivas. Como dijimos en la sección, esto ocurre porque al haber menos prebúsquedas, se reducen los accesos en ese modo, por lo que sube en numerador de la división de la precisión. Por lo que respecta a la cobertura, apreciamos que la mayoría de las configuraciones mostrando mejor precisión tienden a ser el mayor cobertura con el prefetch más agresivo. Siguiendo con el razonamiento en la sección de enteros, es lógico que ocurra esto porque se hacen menos accesos por fallos de L3 gracias a que hay tantos prefetches cuanto más agresivo sea.

	DSCR = 0	DSCR = 66	DSCR = 71	DSCR = 450	DSCR = 455
Bwaves	0,52	0,52	0,52	0,52	0,52
Gamess	0,04	0,07	0,04	0,05	0,02
Milc	0,09	0,19	0,19	0,07	0,07
Zeusmp	0,41	0,51	0,50	0,40	0,39
Gromacs	7,12	10,29	12,36	11,15	5,67
Cactusamd	0,48	0,45	0,46	0,50	0,47
Leslie3d	0,56	0,49	0,48	0,55	0,53
Namd	2,03	3,09	2,25	2,28	1,34
Soplex	0,38	0,96	0,96	0,35	0,32
Povray	0,16	0,26	0,09	0,09	0,03
Gemsfdtd	0,25	0,44	0,43	0,22	0,22
Lbm	0,50	0,50	0,50	0,50	0,49

Tabla 3.6: Precisión del prefetch en aplicaciones de números de coma flotante en las configuraciones del DSCR

	DSCR = 0	DSCR = 66	DSCR = 71	DSCR = 450	DSCR = 455
Bwaves	1,00	1,00	1,00	1,00	1,00
Gamess	0,26	0,18	0,12	0,26	0,27
Milc	0,48	0,24	0,24	0,48	0,49
Zeusmp	0,92	0,58	0,58	0,93	0,93
Gromacs	0,92	0,58	0,74	0,94	0,98
Cactusamd	1,00	0,48	0,49	1,00	1,00
Leslie3d	0,98	0,98	0,98	0,98	0,98
Namd	0,93	0,78	0,82	0,91	0,95
Soplex	0,72	0,48	0,50	0,71	0,74
Povray	0,24	0,06	0,02	0,17	0,26
Gemsfdtd	0,80	0,68	0,68	0,80	0,81
Lbm	1,00	1,00	1,00	1,00	1,00

Tabla 3.7: Cobertura del prefetch en aplicaciones de números de coma flotante en las configuraciones del DSCR

CAPÍTULO 4
Microbenchmark

Significado del microbenchmark

En este estudio, empleamos el término *microbenchmark* como una aplicación auxiliar que se utiliza para realizar un experimento pequeño y controlado, de manera que sepamos de manera teórica qué vamos a obtener, para que a la hora de ejecutarlo en la máquina veamos si los resultados se corresponden con los esperados.

El propósito de éste no es más que contrastar la funcionalidad de los contadores de prestaciones. Muchas veces, los contadores de prestaciones no son precisos y no explicitan el evento que se está midiendo de manera unívoca. A continuación se muestra como ejemplo los contadores de prestaciones para eventos de memoria:

PM_DATA_FROM_MEMORY	La caché de datos del procesador se ha recargado desde la memoria incluyendo la caché local L4, remota o distante, debido a, solamente peticiones de lecturas regulares o peticiones de lecturas regulares más prefetches si MMCR1[16] es 1.
PM_DATA_ALL_FROM	La caché de datos del procesador se ha recargado desde la memoria incluyendola caché L4 local, remota o distante, debido a, solamente peticiones de lecturas regulares o peticiones de lecturas regulares más prefetches si MMCR1[16] es 1.
PM_DATA_FROM_MEM	Datos de caché recargados desde memoria (incluyendo la caché L4)

Se sabe que el registro MMCR1 es un registro de control de modo monitor, pero el problema es que no hay una interfaz que, al igual que *ppc64_cpu*, nos diga el estado de este registro. Por lo tanto, hemos decidido realizar un experimento para determinar si el contador está contando o no los accesos considerados por el prefetch. Además, también queremos ver qué diferencias hay entre estos contadores, si las hay (se aprecia que hay descripciones repetidas), y saber qué evento utilizar.

Scripts y funcionamiento

Para este pequeño experimento hemos utilizado una herramienta del paquete de perf, un pequeño programa que se llama **task**.

```
for dscr in 1 0 66 455 71 450;
do
  frase='ppc64_cpu --dscr=$dscr'
  echo $frase
  echo "#$dscr" >> "res_accuracy.txt"
  echo "LLC" >> "res_accuracy.txt"

  echo "bzip2" >> "res_accuracy.txt"
  ./libpfm-4.7.0/perf_examples/task -e LLC-LOADS,LLC-LOAD-MISSES,PM_MEM_PREF,
      cycles,instructions /home/jacarvi/working_dir/spec_bin/bzip2.ppc64 /home/
      jacarvi/working_dir/CPU2006/401.bzip2/data/all/input/input.combined 200
      >> "res_accuracy.txt"

done
```

Listing 4.1: Script para la ejecución de task del paquete perf

Task tiene un parámetro, **-e**, por el cual se le pasan los eventos que se quieren medir y como segundo parámetro se le pasa el ejecutable, que puede tener sus parámetros también. Todo esto lo ponemos en un fichero para después poder analizarlo sintácticamente y ponerlo en formato CSV. En el listing 4.1 se muestra cómo sería para la ejecución de un benchmark como los que hemos utilizado hasta ahora.

El microbenchmark es un programa en C, cuyo detalle se aprecia en el listing 4.2. Contiene un vector de enteros, cuyo tamaño utilizamos para hacer cálculos sobre los accesos que se van a hacer en el subsistema de memoria. Sabemos que los enteros en esta máquina ocupan 16 bits o 2 bytes. En la variable N, línea 3 del listing 4.2 se define el tamaño que va a tener el vector de enteros, a lo que faltaría multiplica por el tamaño de un integer. Por ejemplo, en la N que tenemos en este listing, $1024 \cdot 1024 \cdot 40 \cdot 32$, resulta ser $5 \cdot 2^{28}$, que si le multiplicamos también los dos bytes de los integers (como se hace en el *malloc* en la línea 17 del listing 4.2) obtenemos $5 \cdot 2^{30}$, que son 5GB. Entonces, si queremos cambiar el tamaño del vector solamente tenemos que modificar N.

```
#include <sys/time.h>

#define N 1024*1024*40*32

main (int argc, char *argv[]) {
  int i, j, k, mem_accesses = 0;
  int *A;
  int nop_ops, mem_ops, acces;
  int iter;
  int curr_array_index = 0;

  if (argc != 4) {
    printf("Error (Numero de arguments = %d). Us prg1 iter nop_ops acces(stride
        )\n", argc);
    return 1;
  }

  A = (int *) malloc (sizeof (int *) * N);

  sleep (5);

  iter = atoi (argv[1]);
  nop_ops = atoi(argv[2]);
  acces = atoi(argv[3]);
  printf("Iter: %d\n", iter);

  for (j=0; j<iter; j++) {
```

```
27      for (k=0; k<N; k+=acces) {
28        A[k]++;
29        mem_accesses += 1;
30      }
31      for (i=0; i<nop_ops; i++) {
32        asm("nop");
33      }
34    }
35    sleep(5);
36    printf("El numero teoric de accessos a memoria es: %d\n", mem_accesses);
37    sleep (5);
38    return 0;
39  }
```

Listing 4.2: Código del microbenchmark.

Planteamiento teórico

Tamaño de bloque

El tamaño de bloque de la caché lo hemos obtenido de manera experimental, debido a la falta de herramientas que nos muestren este tamaño. Para ello, se ha desactivado el prefetch para realizar solo accesos normales. Se ha utilizado el contador de eventos de Perf **LLC-LOAD-MISSES**, que se define como el número de fallos en la caché de L3, que es lo mismo que los accesos que se tendrán que hacer a memoria principal para obtener los datos.

El experimento se ha hecho variando el **stride**, que es la distancia (en bytes) que hay entre los datos accedidos. Por ejemplo, si tenemos stride = 1, estamos accediendo a los datos secuencialmente; si tenemos stride = 4, estamos seleccionando un dato, y el siguiente está cuatro posiciones más alejado del anterior, y así sucesivamente.

Hemos obtenido experimentalmente, en la tabla 4.1, el número de accesos accesos contando las veces que se accede teóricamente al array, que es de 5GB; también los accesos que se hacen a memoria principal con el contador de eventos que hemos mencionado anteriormente; los dividimos entre ellos en la columna de equivalencia, para saber cuantas lecturas al array se hacen por cada acceso a MP. Es decir, este valor indica cuantos elementos se leen del array en caché L3 antes de traer otro bloque. En la equivalencia compensada, multiplicamos la equivalencia por su stride para tener todas las cifras iguales en las filas, y en Tamaño de bloque de caché obtenemos la equivalencia compensada multiplicada por el tamaño en bytes de un entero en este sistema. De esta forma, obtenemos el tamaño de bloque.

Como se puede observar, el tamaño de bloque en bytes, experimentalmente obtenido, sale que es, redondeando a potencias de dos, de **128 bytes**.

Stride	Lecturas del Array	Accesos a MP	Equivalencia	Equivalencia Compensada	Tamaño Int(B)	**Tamaño de Bloque Caché(B)**
1	1342177280	41616027	32,25	32,25	4	129,00
2	671088640	41449987	16,19	32,38	4	129,52
4	335544320	41521190	8,08	32,32	4	129,30
8	167772160	41509669	4,04	32,33	4	129,33
16	83886080	41539547	2,02	32,31	4	129,24
32	41943040	41515081	1,01	32,33	4	129,32

Tabla 4.1: Cálculos para el tamaño de bloque

Resultados prácticos obtenidos

Diferencias entre contadores

Para un vector de 5GB de enteros, hemos hecho dos tipos de medidas. En la tabla 4.2, la columna Prefetch OFF se ha hecho con DSCR = 1, que significa que no hay prefetch (a pesar de que en PM_MEM_PREF haya una cifra mayor que 0, es una cifra menospreciable). La columna de Prefetch ON se ha hecho con el prefetch que viene por defecto.

Contador	Prefetch OFF	Prefetch ON (Default)
PM_DATA_FROM_MEM	39381023	53303
PM_DATA_FROM_MEMORY	39318878	54990
PM_DATA_ALL_FROM_MEMORY	39352290	60283
LLC-LOAD-MISSES	42241320	235541
PM_DATA_FROM_L3MISS	42234162	234989
PM_MEM_PREF	2100	80858202

Tabla 4.2: Comparación entre contadores de eventos

Como podemos observar, los contadores que tienen que ver con las lecturas que hay en la memoria, que son PM_DATA_FROM_MEM, PM_DATA_FROM_MEMORY y PM_DATA_ALL_FROM_MEMORY, miden prácticamente lo mismo. Las diferencias entre ellos son pequeñas, sobretodo si nos fijamos en la columna en la que no hay prefetch. En los contadores de los fallos de L3, LLC-LOAD-MISSES y PM_DATA_FROM_L3MISS, vemos que también miden prácticamente lo mismo. Entonces, queda claro que las diferencias entre los contadores del mismo grupo son mínimas.

En lo que sí hay una diferencia es entre los fallos de LLC y los accesos a memoria principal. Deberían ser lo mismo teóricamente, pero son más numerosos los fallos en LLC, lo cual se debe a que pueden haber varios fallos en el mismo bloque de la caché LLC, mientras que en memoria principal solamente representaría un fallo. Y la otra diferencia, y la más relevante en esta sección, es que como podemos observar en la fila de los prefetches en memoria principal, cuando el prefetch está activado, hay una cifra que es muy elevada respecto de las demás. Esto hace que quede claro que **los contadores de cargas desde memoria no miden los accesos en prefetch**, porque no hay ninguna cifra de las cargas que supere la de los prefetches. Si los contadores de las loads desde MP los estuvieran contado, tendría que ser una cifra superior a los prefetches, ya que se sumarían las loads corrientes.

Cobertura y precisión

Como hemos explicado en los apartados 3.3.2 y 3.4.2, que versan sobre la precisión y la cobertura, vamos a aplicar las fórmulas de la cobertura y de la precisión para ver si en nuestro experimento controlado también ocurre lo que en las aplicaciones de la *benchmark suite* SPEC. Tomamos un vector de 1 GB, con stride 1, y obtenemos el número de accesos a memoria principal teóricamente sin tener activado el prefetch. $\frac{1GB}{128B} = \frac{2^{30}}{2^7} = 8388608$, que aproximadamente es 8,4 millones de accesos. Obtenido experimentalmente este dato con el contador de fallos de L3 PM_DATA_FROM_L3MISS, vemos que es coincidente, pues tenemos **8441760** accesos, que también se acerca a la cifra redondeada. Recordemos que los fallos de L3 se acercan mucho al número de accesos a memoria principal; no estamos usando ésta porque al ser un vector que accede a posiciones de memoria contiguas, muchas veces accede al mismo bloque de memoria, por lo que la memoria resultaria una cifra inferior. Entonces, sabiendo este dato, aplicamos las fórmulas correspondientes de la precisión 3.5 y de la cobertura 3.7.

	DSCR = 0	DSCR = 66	DSCR = 71	DSCR = 450	DSCR = 455
PM_DATA_FROM_L3MISS	44002	48087	46799	40262	40297
PM_MEM_PREF	219393	59522	61700	357030	444312
Precisión	38,28	141,02	136,06	23,53	18,91
Cobertura	0,99	0,99	0,99	0,99	0,99

Tabla 4.3: Precisión y cobertura en el microbenchmark de 1 GB.

Para corroborar las medidas de la tabla 4.3, tomamos ahora un vector de 5 GB. Los accesos teóricos son 41943040 y los prácticos obtenidos con PM_DATA_FROM_L3MISS son **42251990**. Como ocurría antes, los valores experimentales no coinciden exactamente con los teóricos, pero están muy cerca. Y así, vemos en la tabla 4.4 la cobertura y la precisión, que tienen el mismo comportamiento que con el vector de 1 GB:

	DSCR = 0	DSCR = 66	DSCR = 71	DSCR = 450	DSCR = 455
PM_DATA_FROM_L3MISS	244896	267756	2677266	52873	232270
PM_MEM_PREF	481950	480410	1218338	1690652	2212378
Precisión	32,31	81,64	81,91	23,18	17,80
Cobertura	0,99	0,99	0,99	0,99	0,99

Tabla 4.4: Precisión y cobertura en el microbenchmark de 5 GB.

Al igual que con las aplicaciones anteriores, la precisión experimenta una bajada en las configuraciones en las que la urgencia es alta; es mejor cuando la urgencia es menor, porque se hacen menos accesos en prefetch, lo que mejora la precisión. Por otro lado, la cobertura es tan alta que no se puede apreciar la diferencia entre las configuraciones. Esto es porque el número de fallos de L3 es muy bajo, lo cual es buena señal en este experimento controlado.

CAPÍTULO 5
Conclusiones

El prefetch provoca distintos tipos de reacciones en las aplicaciones. Provoca tanto un beneficio en las prestaciones muy alto, un beneficio pequeño, indiferencia y empeoramiento en el tiempo de ejecución de las aplicaciones. Esto se debe a los fallos en las cachés que conllevan accesos a la memoria principal, las lecturas, cuya duración es muy alta comparada con las duraciones de las cachés; si el prefetch ahorra accesos a memoria principal, mejorará el rendimiento, que se mide a través del IPC, y la comparativa se hace con speedup.

La cobertura y la precisión las hemos utilizado para verificar que los accesos a memoria y los prefetches resultaban una cifra razonable, lo cual ha sido así: el análisis de la precisión indica que son más precisas aquellas aplicaciones que hacen menos prebúsquedas, porque se hacen más accesos pero el número de prefetches se reduce considerablemente. Por otro lado, la cobertura es mayor cuando mayor es la agresividad del prefetch, porque el número de accesos en memoria principal se reduce considerablemente.

También se ha implementado el microbenchmark para ver si con un experimento controlado obteníamos lo esperado y concordaba con lo obtenido con las aplicaciones del paquete SPEC, que así fue. La precisión ha sido mayor en las aplicaciones con prefetch menos agresivo, pero la cobertura ha sido tan alta en todos los casos que apenas se ha podido apreciar. También nos ha servido para comprobar que los eventos que miden los accesos hacen lo que dicen hacer, que, desafortunadamente, no siempre ocurre. Sin embargo, hemos calculado los valores teóricos y hemos verificado que coinciden con los prácticos.

Bibliografía

[1] Documentación de la Wikipedia sobre el cache prefetching.
Consultado el 5 de Marzo de 2017.
https://en.wikipedia.org/wiki/Cache_prefetching

[2] Wiki oficial de IBM, sobre los usos frecuentes del registro DSCR.
Consultado el 28 de Abril de 2017.
https://www.ibm.com/developerworks/community/wikis/home?lang=en#!/wiki/W51a7ffcf4dfd_4b40_9d82_446ebc23c550/page/A+Proper+Use+of+Data+Stream+Control+Register+of+POWER+Processors

[3] Página oficial de la *benchmark suite* SPEC CPU® 2006.
Consultado el 30 de Abril de 2017.
https://www.spec.org/cpu2006/.

[4] Archivo PDF de IBM sobre el Monitor del contador de prestaciones, Qi L. (2009).
Consultado el 19 de Mayo de 2017.
https://www.ibm.com/developerworks/aix/library/au-counteranalyzer/au-counteranalyzer-pdf.pdf

[5] Wiki de Linux sobre la utilidad *perf*.
Consultado el 18 de Mayo de 2017.
https://perf.wiki.kernel.org/index.php/Main_Page

[6] Página oficial de IBM donde se describen los eventos.
Consultado el 2 de Octubre de 2017.
https://www.ibm.com/support/knowledgecenter/SSFK5S_2.2.0/com.ibm.cluster.pedev.v2r2.pedev100.doc/bl7ug_power8metrics.htm

[7] Características técnicas del IBM Power8.
Consultado el 6 de Septiembre de 2017.
http://www.7-cpu.com/cpu/Power8.html

[8] Sitio oficial de Oprofile con una lista de eventos para IBM Power8.
Consultado el 12 de Julio de 2017.
http://oprofile.sourceforge.net/docs/ppc64-power8-events.php