Análisis de la velocidad de memoria en diferentes dispositivos

Diego Alejandro García Barajas
Escuela de Ingeniería de Sistemas
Universidad Industrial de Santander
Bucaramanga, Colombia
dagb170422@gmail.com

Jefferson Holguín Ferro
Escuela de Ingeniería de Sistemas
Universidad Industrial de Santander
Bucaramanga, Colombia
jefferrholguin@gmail.com

Cristian Manuel Hernández Delgado
Escuela de Ingeniería de Sistemas
Universidad Industrial de Santander
Bucaramanga, Colombia
cristianmanuel2304@hotmail.com

Manuela Alejandra García Valbuena
Escuela de Ingeniería de Sistemas
Universidad Industrial de Santander
Bucaramanga, Colombia
mg2472493@gmail.com

Santiago Gómez Villarreal
Escuela de Ingeniería de Sistemas
Universidad Industrial de Santander
Bucaramanga, Colombia
santiagogv0506@gmail.com

Resumen

Este artículo presenta un programa desarrollado en C utilizando MS Visual Studio para evaluar el comportamiento de sistemas de memoria. El programa se enfoca en lograr una sincronización precisa y luego realizar accesos a memoria que abarcan diferentes niveles de jerarquía de caché. La metodología incluye la medición precisa del tiempo de CPU utilizando utilidades estándar, seguido de bucles anidados para lectura y escritura en memoria con diferentes tamaños y pasos de caché. Los resultados se calculan en términos de tiempo de acceso y se generan en formato .csv para su análisis en hojas de cálculo.

Abstract

This paper introduces a C program developed in MS Visual Studio to evaluate memory system behavior. The program aims to achieve precise synchronization and then traverse memory accesses spanning various cache hierarchy levels. The methodology includes precise CPU time measurement using standard utilities, followed by nested loops for memory read and write operations with varying cache sizes and strides. Results are computed in terms of access times and outputted in .csv format for spreadsheet analysis.

I. Introducción

En el estudio de la arquitectura de computadoras, comprender el comportamiento y rendimiento de los sistemas de memoria es crucial para el diseño eficiente de hardware y software. El libro "Computer Architecture: A Quantitative Approach" de Patterson y Hennessy, específicamente en su quinta edición, dedica un análisis profundo a este tema. Basándose en los casos de estudio y ejercicios desarrollados por Norma P. Jouppi, Naveen Muralimanohar y Sheng LI, uno de los estudios más reveladores es el Caso de Estudio 2, titulado "Poniendo todo junto: sistemas de memoria altamente paralelos" (página 133).

Este caso de estudio presenta un programa en lenguaje C, diseñado para ejecutarse en MS Visual Studio, cuyo propósito es evaluar el comportamiento de un sistema de memoria. El programa se centra en la sincronización precisa y en la invocación de diferentes niveles de jerarquía de memoria, desde las cachés más pequeñas hasta las más grandes. Este análisis se realiza mediante la lectura y escritura en la memoria, permitiendo observar cómo el sistema maneja distintos tamaños de caché y diferentes pasos en el recorrido de la memoria.

El programa se puede dividir en tres partes principales:

1. Sincronización Precisa: La primera parte del programa utiliza una utilidad estándar para obtener medidas precisas del tiempo de CPU usuario. Esto es fundamental para asegurar que los tiempos medidos sean lo más exactos posible, minimizando cualquier inexactitud en la sincronización del reloj.
2. Bucle Anidado para Accesos a Memoria: La segunda parte consiste en un bucle anidado que realiza lecturas y escrituras en la memoria. Este bucle varía los tamaños de los pasos y las cachés, lo que permite evaluar el rendimiento del sistema de memoria en diferentes configuraciones. Este método es crucial para entender cómo el acceso a la memoria y el tamaño de la caché afectan el rendimiento general del sistema.
3. Cálculo de Tiempos de Sobrecarga: La tercera parte del programa calcula los tiempos de sobrecarga del bucle anidado. Esto se logra restando los tiempos totales medidos para determinar cuánto tiempo realmente toman los accesos a la memoria, excluyendo cualquier sobrecarga adicional introducida por el bucle mismo.

A través de este programa, los estudiantes y profesionales de la arquitectura de computadoras pueden obtener una visión detallada de cómo las configuraciones de memoria afectan el rendimiento. La medición precisa del tiempo de acceso a diferentes niveles de caché proporciona datos valiosos para optimizar tanto el hardware como el software, logrando sistemas más eficientes y de mayor rendimiento.

II. Contenido

A. Compilando, ejecutando el código para evaluar memoria en C.

En este apartado se realiza una descripción detallada del código y su funcionamiento, además de observaciones que se encontraron en tiempo de ejecución, posteriormente en el siguiente punto, se muestran y analizan los resultados al compilar el código.

1. Se define el máximo cache y se declara un array (que se va a usar para medir)

```
#define ARRAY_MIN (1024) /* 1/4 smallest cache */
#define ARRAY_MAX (4096*4096) /* 1/4 largest cache */
int x[ARRAY_MAX]; /* array going to stride through */
```

2. Se genera una función para obtener los segundos. La implementación varía dependiendo de si se usa Ms visual studio o no. En este caso se está usando el compilador GCC, así que se prescinde de visual. La estructura timeval viene de la librería <sys/time.h> y se usa para guardar los segundos o milisegundos. gettimeofday(&tv, NULL) se usa para tener el tiempo actual del sistema. La función retorna la suma de los segundos más los milisegundos convertidos a segundos, ósea la cantidad total de segundos.

```
#ifdef _MSC_VER
double get_seconds() { /* routine to read time in seconds */
  __time64_t ltime;
  _time64(&ltime);
  return (double) ltime;
}
#else
double get_seconds() {
  struct timeval tv;
  gettimeofday(&tv, NULL);
  return tv.tv_sec + tv.tv_usec / 1e6;
}
#endif
```

3. Esta función se encarga de generar los labels. La función recibe como entrada la cantidad de bytes y dependiendo de esta, va cambiando el sufijo a imprimir (B, K , M G).

```
int label(int i) { /* generate text labels */
  if (i < 1e3)
    printf("%1dB,", i);
  else if (i < 1e6)
    printf("%1dK,", i / 1024);
  else if (i < 1e9)
    printf("%1dM,", i / 1048576);
```

```
  else
    printf("%1dG,", i / 1073741824);
  return 0;
}
```

4. Dependiendo de si se está usando Ms Visual Studio o GCC la función main cambia. Ya una vez declarada, se declaran las variables a usar durante la ejecución del programa. Luego con el bucle for se imprimirán las respectivas etiquetas. Estas etiquetas actúan como cabecera del csv final.

```
#ifdef _MSC_VER
int _tmain(int argc, _TCHAR* argv[]) {
#else
int main(int argc, char* argv[]) {
#endif
  int register nextstep, i, index, stride;
  int register csize;
  double steps, tsteps;
  double loadtime, lastsec, sec0, sec1, sec; /* timing variables */

  /* Initialize output */
  printf(" ,");
  for (stride = 1; stride <= ARRAY_MAX / 2; stride = stride * 2)
    label(stride * sizeof(int));
  printf("\n");
```

5. Este bucle es la parte más importante del código pues es el que se usará para obtener los resultados. Al inicio de cada iteración del bucle principal se llama la función label para generar la respectiva etiqueta (se puede interpretar como la etiqueta correspondiente para cada fila del csv) que representa el tamaño del caché. El bucle va desde el mínimo al máximo cache, duplicando el caché a utilizar en cada iteración. En el bucle interno se trabajan con punteros, para simular el acceso a memora, cada iteración el valor a apuntar a la dirección actual más el paso o "stride", siendo como un desplazamiento en la memoria. Stride va hasta la mitad del tamaño actual del caché.

```
for (csize = ARRAY_MIN; csize <= ARRAY_MAX; csize = csize * 2) {
  label(csize * sizeof(int)); /* print cache size this loop */
  for (stride = 1; stride <= csize / 2; stride = stride * 2) {
    /* Lay out path of memory references in array */
    for (index = 0; index < csize; index = index + stride)
      x[index] = index + stride; /* pointer to next */
    x[index - stride] = 0; /* loop back to beginning */
```

6. Este bloque de código está diseñado para medir el tiempo necesario para acceder a la memoria. Primero, se espera a que el temporizador se reinicie para asegurar mediciones precisas del tiempo. Luego, se ejecuta un bucle que simula el acceso a memoria durante aproximadamente 20 segundos. En cada iteración de este bucle, se accede secuencialmente a los elementos del arreglo x utilizando el índice nextstep, que se incrementa según el tamaño del paso (stride). Este proceso se repite hasta alcanzar los 20 segundos de tiempo total medido ((sec1 - sec0) < 20.0). Después de recoger los datos, se calcula el tiempo total utilizado (sec1 - sec0), proporcionando así una medida del rendimiento del acceso a memoria para la configuración actual de tamaño de caché (csize) y paso (stride).

```
/* Wait for timer to roll over */
lastsec = get_seconds();
do sec0 = get_seconds(); while (sec0 == lastsec);
/* Walk through path in array for twenty seconds */
/* This gives 5% accuracy with second resolution */
steps = 0.0; /* number of steps taken */
nextstep = 0; /* start at beginning of path */
sec0 = get_seconds(); /* start timer */
do { /* repeat until collect 20 seconds */
```

```
  for (i = stride; i != 0; i = i - 1) { /* keep samples same */
    nextstep %= csize;
    do nextstep = x[nextstep]; /* dependency */
    while (nextstep != 0);
  }
  steps = steps + 1.0; /* count loop iterations */
  sec1 = get_seconds(); /* end timer */
} while ((sec1 - sec0) < 20.0); /* collect 20 seconds */
sec = sec1 - sec0;
```

7. Se inicializan variables para medir el tiempo (tsteps, index, sec0, sec1) y se inicia el temporizador (sec0 = get_seconds()). Luego, se ejecuta un bucle que repite un número de iteraciones igual al número de pasos tomados anteriormente (steps). Dentro de este bucle, se ejecuta otro bucle que simula un acceso vacío a la memoria para igualar la carga de trabajo del bucle principal. Se suma uno al contador de iteraciones (tsteps = tsteps + 1.0;) y se registra el tiempo al finalizar el bucle (sec1 = get_seconds();). Después de completar todas las iteraciones necesarias, se calcula el tiempo de ejecución total, restando el tiempo de sobrecarga medido previamente (sec = sec - (sec1 - sec0)). Finalmente, se calcula el tiempo de carga (loadtime) en nanosegundos por acceso a memoria, y se imprime como csv.

```
/* Repeat empty loop to loop subtract overhead */
tsteps = 0.0; /* used to match no. while iterations */
index = 0; /* start at beginning of path */
sec0 = get_seconds(); /* start timer */
do { /* repeat until same no. iterations as above */
  for (i = stride; i != 0; i = i - 1) { /* keep samples same */
    index %= csize;
    do index = index + stride;
    while (index < csize);
  }
  tsteps = tsteps + 1.0;
  sec1 = get_seconds(); /* - overhead */
} while (tsteps < steps); /* until = no. iterations */
sec = sec - (sec1 - sec0);
loadtime = (sec * 1e9) / (steps * csize);
/* write out results in .csv format for Excel */
printf("%4.1f,", (loadtime < 0.1) ? 0.1 : loadtime);
```

B. Resultados y gráficas

1. 16 GB de RAM DDR4 en Ryzen 3 3200g

A partir de la ejecución del código anterior en c, se generan una serie de resultados de tiempo (en ns) que demora la memoria en procesar cierta cantidad de información desde los 4 bits hasta 64 Mb, usando diversos tamaños de memoria desde los 4Kb hasta 64Mb. Toda esta información es entregada en formato CVS y se presenta a continuación en la tabla 1 en una forma más fácil de analizar.

Tabla 1: Resultados de ejecución en el primer dispositivo.

Size	4E	8E	16	32	64	12	25	51	1K	2K	4K	8K	16	32	64	12	25	51	1M	2M	4M	8M	16	32	64
4K	1,3	1,2	1,2	1,2	1,2	1,2	1,3	1,3	1,3	1,2															
8K	1,4	1,3	1,3	1,3	1,2	1,2	1,3	1,3	1,3	1,3	1,2														
16K	1,3	1,3	1,3	1,3	1,3	1,2	1,2	1,3	1,3	1,3	1,3	1,2													
32K	1,3	1,3	1,3	1,3	1,3	1,3	1,3	1,3	1,6	1,3	1,3	1,3	1,2												
64K	1,3	1,3	1,3	1,3	1,3	1,3	1,3	1,3	1,3	2,5	2,6	1,3	1,3	1,2											
128K	1,3	1,3	1,3	1,3	1,4	1,4	1,4	1,4	1,4	2,6	2,5	2,6	1,3	1,3	1,2										
256K	1,3	1,3	1,3	1,3	1,3	1,8	1,7	1,8	2,1	2,9	3,1	2,5	3,6	1,3	1,3	1,2									
512K	1,3	1,3	1,3	1,3	1,4	1,6	1,6	1,6	1,8	2,8	3,2	2,8	4,1	5,3	1,3	1,3	1,2								
1M	1,3	1,3	1,3	1,3	1,6	1,9	2,2	2,6	2,9	4,8	4,9	5,0	9,5	9,6	9,3	1,3	1,3	1,2							
2M	1,3	1,3	1,3	1,3	1,6	1,9	2,2	2,6	2,9	4,9	4,9	5,0	11,5	9,5	9,5	9,4	1,3	1,3	1,2						
4M	1,3	1,3	1,3	1,4	1,8	2,1	2,4	3,0	3,4	5,3	5,3	5,5	12,1	12,3	9,5	9,5	9,5	1,3	1,3	1,2					
8M	1,3	1,4	1,6	2,9	5,5	8,5	12,6	15,3	17,6	29,0	35,4	34,8	99,4	99,7	98,0	91,3	88,1	9,4	1,3	1,3	1,2				
16M	1,3	1,4	1,6	2,9	5,4	8,6	12,7	15,6	17,8	30,0	36,6	36,5	104,7	104,1	103,2	103,1	91,8	90,0	9,4	1,3	1,3	1,2			
32M	1,3	1,4	1,7	2,8	5,5	8,7	12,8	15,8	17,9	30,0	36,7	36,8	105,7	106,6	105,7	105,4	105,1	93,2	88,9	9,5	1,3	1,3	1,2		
64M	1,3	1,4	1,6	2,7	5,3	8,7	12,9	15,6	17,9	30,1	36,7	36,8	104,9	105,7	104,6	105,0	103,6	100,4	83,2	77,2	9,1	1,3	1,3	1,2	

Es importante resaltar que los campos vacíos se dan por que no se pueden procesar una cantidad de datos de N bits con una memoria de M bits donde M≤N.

Para entender estos datos se pueden representar mediante una gráfica de tiempo versus cantidad de información procesada (ver la Ilustración 1). Lo primero a destacar es que los tiempos de ejecución siguen el principio de localidad que nos dice que entre menor sea el nivel del caché o más cercano se encuentre al procesador, más rápida será la ejecución (ver la Ilustración 2).

Ilustración 1: Grafico de la Tabla 1.

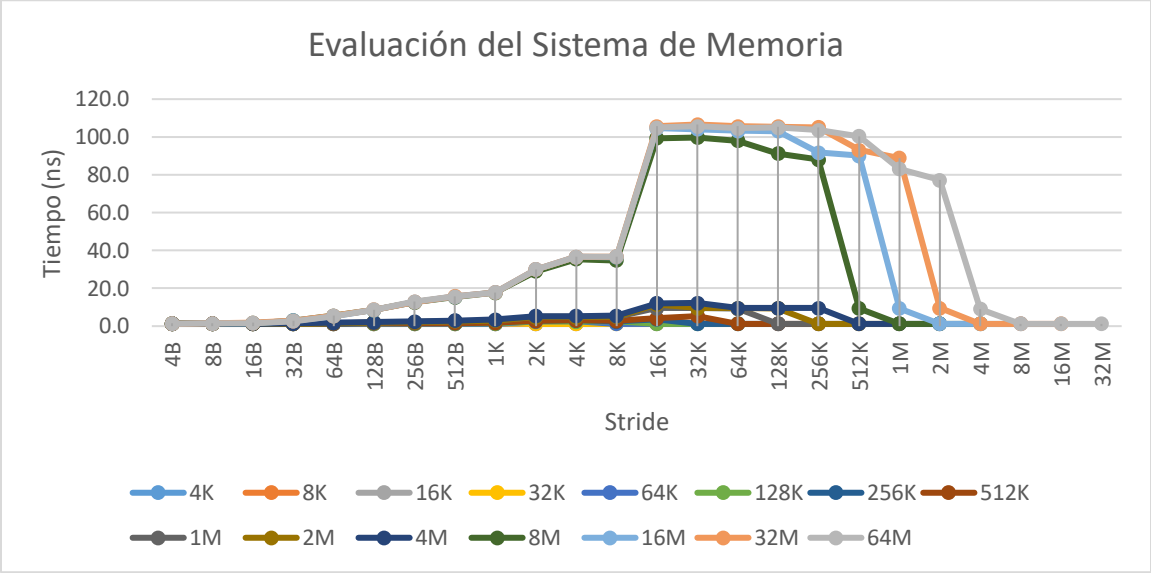
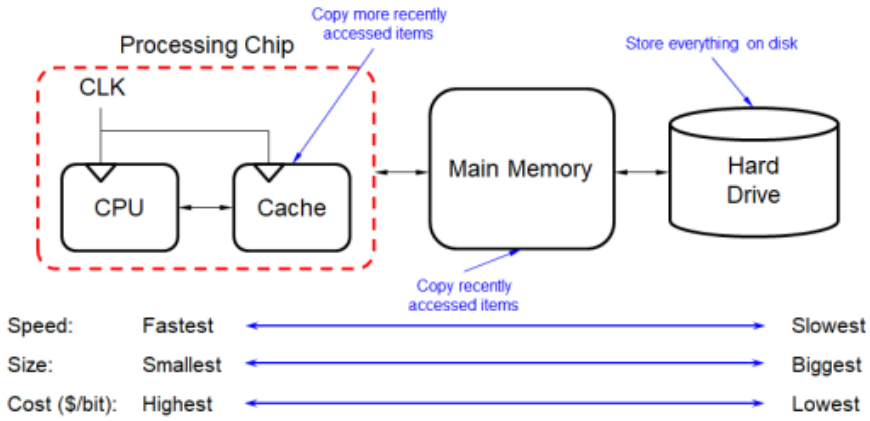


Ilustración 2: Principio de localidad.[1]



Para este procesador, se observa un comportamiento rápido en cuanto al manejo de la información almacenada en caché y los accesos directos a la memoria principal, cuando la cantidad de información a procesar es mayor a la capacidad del caché asignado. Para este equipo en configuración predeterminada del fabricante AMD el mayor tiempo registrado fue de 106.6 ns, procesando 32 Kb con un espacio de memoria de 32Mb.

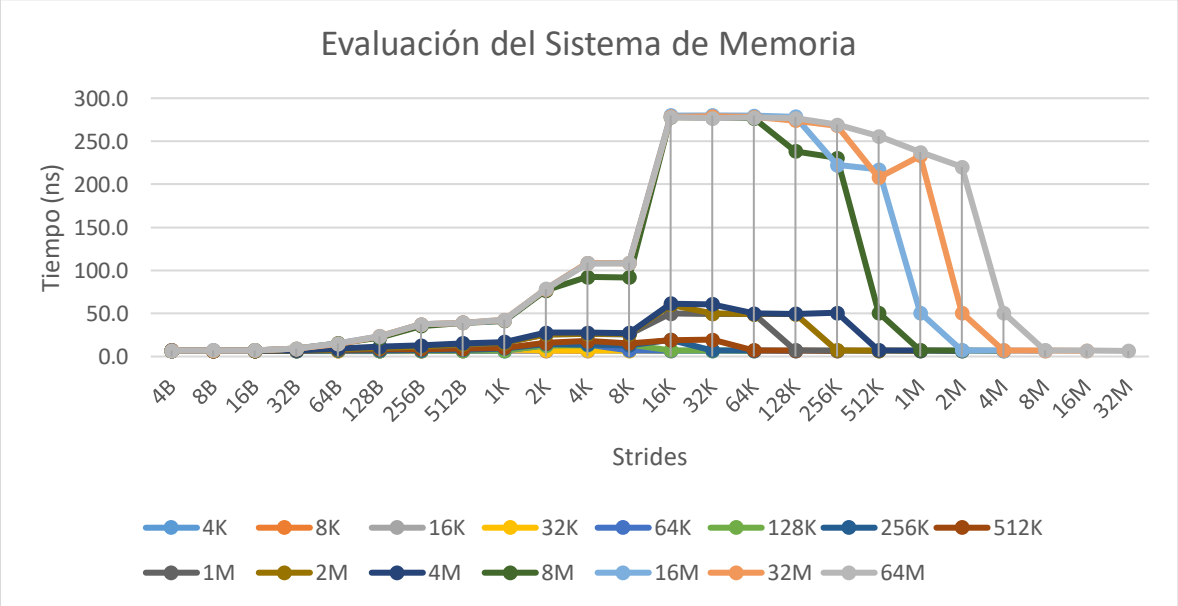
1. 8 GB de RAM DDR4 en un AMD 3020e

Por otra parte en otro dispositivo AMD también en configuración predeterminada se ha ejecutado el mismo código, cuya salida se encuentra registrada en la tabla 2. Es importante realizar ciertas observaciones respecto al equipo puesto que sus componentes son más antiguos y lleva mucho más tiempo en uso, algo que se evidencia en los resultados de la tabla pues el tiempo máximo obtenido es de 280 ns, más del doble del anterior dispositivo, también para la cantidad de 32 Kb pero con un tamaño de memoria de solo 16 Mb.

Tabla 2: Salida de la ejecución del programa para el segundo dispositivo.

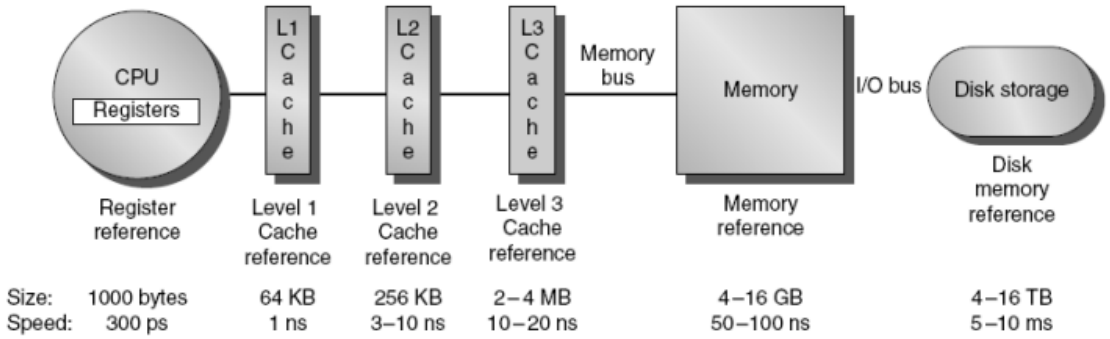
Size	4E	8E	16	32	64	12	25	51	1K	2K	4K	8K	16	32	64	12	25	51	1M	2M	4M	8M	16	32	6
4K	6,9	6,7	6,7	6,5	6,3	6,4	6,9	7,0	6,9	6,4															
8K	7,0	6,8	6,8	6,7	6,6	6,4	6,5	7,0	7,0	6,9	6,5														
16K	6,9	6,9	6,8	6,7	6,7	6,6	6,4	6,6	7,0	7,0	6,8	6,4													
32K	6,9	6,9	6,9	6,9	6,9	6,8	6,8	6,7	8,5	7,0	7,0	6,8	6,4												
64K	6,9	6,9	6,9	6,9	7,0	6,9	6,9	6,9	6,9	13,0	13,9	7,0	6,7	6,7											
128K	6,9	6,9	6,9	6,9	7,0	6,9	6,9	6,9	6,9	12,8	13,1	14,0	7,0	6,8	6,5										
256K	6,9	6,9	6,9	6,9	7,1	7,2	7,5	8,1	9,3	14,5	13,9	13,2	19,3	7,0	6,9	6,5									
512K	6,9	6,9	6,7	7,0	7,8	8,5	8,8	9,2	10,0	15,6	18,0	15,4	19,2	19,6	7,0	6,9	6,3								
1M	7,0	6,9	6,9	7,0	8,3	10,2	11,7	14,0	15,6	25,6	26,0	25,9	50,0	49,5	49,0	7,0	6,9	6,4							
2M	6,9	6,9	6,9	7,0	8,4	10,3	11,8	14,3	15,7	25,6	26,1	26,1	59,9	50,0	49,3	49,0	7,0	6,9	6,4						
4M	6,9	6,9	7,0	7,3	9,0	11,1	12,8	15,5	16,9	27,5	27,7	27,4	61,3	60,7	50,0	49,4	50,8	7,0	6,9	6,4					
8M	6,9	6,9	7,3	9,0	15,3	21,6	35,4	39,1	41,2	76,8	92,3	91,9	278,0	277,3	276,6	238,2	230,5	50,6	7,0	6,8	6,5				
16M	6,9	7,0	7,3	9,0	15,0	23,3	37,3	39,3	42,3	78,3	108,4	108,5	279,9	280,0	279,5	278,5	222,3	217,2	50,7	7,0	6,9	6,4			
32M	6,9	6,9	7,3	9,0	14,8	23,4	37,4	39,2	42,5	78,3	108,8	108,3	278,6	279,0	278,0	274,2	268,1	207,9	233,1	50,5	7,0	6,9	6,5		
64M	6,9	7,0	7,3	9,0	14,9	23,5	37,4	39,1	42,5	78,4	108,3	108,2	277,9	276,7	277,4	276,5	269,3	255,9	237,3	220,2	50,6	7,0	6,8	6,4	

Ilustración 3: Gráfico de la Tabla 2



Es posible observar que el intervalo de 16Kb a 2Mb presenta los tiempos más altos de retardo en procesamiento de la información. Estos retardos nos llevan a analizar la Jerarquía de memoria (ver la ilustración 4).

Ilustración 4: Jerarquía de memoria.[2]



Dado el comportamiento de las gráficas, es notorio que el nivel 3 de caché representa la mayor demora, pero es posible que en la constitución de los dispositivos no se encuentre este nivel, por lo cual esta cantidad de información es enviada directamente a la RAM, lo cual implica menor velocidad en el manejo de la información. Sin embargo, con grandes cantidades de información se observa que los tiempos de procesamiento no son tan altos.

C. Tamaño Total y Tamaño del Bloque de Caché de Segundo Nivel.

El tamaño de la caché de segundo nivel (L2) y su impacto se pueden inferir de las curvas de tiempo de acceso a la memoria en relación con el stride y el tamaño del bloque:

Identificación del Tamaño del Bloque de Caché de Segundo Nivel:

- En ambas gráficas, los tiempos de acceso empiezan a aumentar significativamente cuando el tamaño del bloque excede un cierto umbral, indicando que los datos están saliendo de la caché y están siendo accedidos desde la memoria principal.
- Para el Ryzen 3 3200g, el aumento significativo de tiempo de acceso ocurre aproximadamente a partir de bloques de 64K, lo que sugiere que la caché L2 es capaz de manejar bloques menores a este tamaño eficientemente.
- Para el AMD 3020e, el aumento significativo también comienza alrededor de los 64K.

Esto sugiere que el tamaño del bloque de caché de segundo nivel para ambos procesadores es aproximadamente 64 KB.

¿Qué significa esto?

- Caché L2 de 64 KB: Significa que la caché de segundo nivel puede almacenar hasta 64 KB de datos. Cuando se realizan accesos a memoria que caben dentro de este tamaño, los tiempos de acceso son muy bajos (indicando que los datos son accesibles rápidamente desde la caché).
- Impacto en Rendimiento: Accesos a datos mayores a 64 KB experimentarán tiempos de acceso mayores, ya que implican acceder a la memoria principal en lugar de la caché, lo cual es mucho más lento.

D. Tamaño de la Memoria Principal.

- Ryzen 3 3200g: Según la información proporcionada, tiene 16 GB de RAM DDR4.
- AMD 3020e: Según la información proporcionada, tiene 8 GB de RAM.

E. Modificaciones al código suministrado e importancia de conocer las características de la cache.

Para medir las características de la caché de instrucciones, se podría realizar algunas modificaciones en el código original, que mide el rendimiento de la caché de datos. Las modificaciones que podemos hacer consisten en:

1. Separar las funciones para medir tiempos de carga de datos e instrucciones añadiendo funciones que ejecuten un conjunto de instrucciones repetitivamente y midan el tiempo de ejecución de dichas instrucciones.
2. Modificar la estructura del bucle principal añadiendo un nuevo bucle que mida el rendimiento de la caché de instrucciones de manera similar a cómo se mide el rendimiento de la caché de datos.
3. Medir el tiempo de ejecución de bloques de instrucciones específicas creando bloques de instrucciones que simulen diferentes tamaños de caché y medir su tiempo de ejecución.

Teniendo esto en cuenta el código modificado quedaría de la siguiente manera:

```
#include <tchar.h>
#include <locale.h>
#include <stdio.h>
#include <time.h>
#include <sys/time.h>

#define ARRAY_MIN (1024)
#define ARRAY_MAX (4096*4096)
int x[ARRAY_MAX]; /

#ifdef _MSC_VER
double get_seconds() {
  __time64_t ltime;
  _time64(&ltime);
  return (double) ltime;
}
#else
double get_seconds() {
  struct timeval tv;
  gettimeofday(&tv, NULL);
  return tv.tv_sec + tv.tv_usec / 1e6;}
#endif
int label(int i) {
  if (i < 1e3) printf("%1dB,", i);
  else if (i < 1e6) printf("%1dK,", i / 1024);
  else if (i < 1e9) printf("%1dM,", i / 1048576);
  else printf("%1dG,", i / 1073741824);
  return 0;
}
void dummy_function(int steps) {
  volatile int dummy = 0;
  for (int i = 0; i < steps; ++i) {
    dummy += i;
  }
}
#ifdef _MSC_VER
int _tmain(int argc, _TCHAR* argv[]) {
#else
int main(int argc, char* argv[]) {
#endif
```

```
  int register nextstep, i, index, stride;
  int register csize;
  double steps, tsteps;
  double loadtime, lastsec, sec0, sec1, sec;

  printf(" ,");
  for (stride = 1; stride <= ARRAY_MAX / 2;
stride = stride * 2)
    label(stride * sizeof(int));
  printf("\n");
  for (csize = ARRAY_MIN; csize <=
ARRAY_MAX; csize = csize * 2) {
    label(csize * sizeof(int)); /* print cache size this
loop */
    for (stride = 1; stride <= csize / 2; stride = stride
* 2) {
      /* Lay out path of memory references in array
*/
      for (index = 0; index < csize; index = index +
stride)
        x[index] = index + stride; /* pointer to next */
      x[index - stride] = 0; /* loop back to beginning
*/
      lastsec = get_seconds();
      do sec0 = get_seconds(); while (sec0 ==
lastsec);

      steps = 0.0;
      nextstep = 0;
      sec0 = get_seconds();
      do {
        for (i = stride; i != 0; i = i - 1) {
          nextstep %= csize;
          do nextstep = x[nextstep];
          while (nextstep != 0);
        }
        steps = steps + 1.0;
        sec1 = get_seconds();
      } while ((sec1 - sec0) < 20.0);
      sec = sec1 - sec0;
      tsteps = 0.0;
      index = 0;
      sec0 = get_seconds();
      do {
        for (i = stride; i != 0; i = i - 1) {
          index %= csize;
          do index = index + stride;
          while (index < csize);
        }
        tsteps = tsteps + 1.0;
        sec1 = get_seconds();
      } while (tsteps < steps);
      sec = sec - (sec1 - sec0);
      loadtime = (sec * 1e9) / (steps * csize);
      /* write out results in .csv format for Excel */
      printf("%4.1f,", (loadtime < 0.1) ? 0.1 :
loadtime);
    }
    printf("\n");
  }
  printf("Instruction Cache,\n");
  for (csize = ARRAY_MIN; csize <=
ARRAY_MAX; csize = csize * 2) {
    label(csize * sizeof(int));
    for (stride = 1; stride <= csize / 2; stride = stride
* 2) {
      steps = 0.0;
      sec0 = get_seconds();
      do {
        for (i = 0; i < stride; ++i) {
          dummy_function(csize);
        }
        steps = steps + 1.0;
        sec1 = get_seconds();
      } while ((sec1 - sec0) < 20.0);
      sec = sec1 - sec0;
      tsteps = 0.0;
      sec0 = get_seconds();
      do {
        for (i = 0; i < stride; ++i) {
        }
        tsteps = tsteps + 1.0;
        sec1 = get_seconds();
      } while (tsteps < steps);
      sec = sec - (sec1 - sec0);
      loadtime = (sec * 1e9) / (steps * csize);
      printf("%4.1f,", (loadtime < 0.1) ? 0.1 :
loadtime);
    }
    printf("\n");
  }
  char varfin[100];
  fgets(varfin, 100, stdin);
  return 0;
}
```

Función dummy_function:

Esta función se utiliza para ejecutar un bloque de instrucciones repetitivas que simulan el uso de la caché de instrucciones. Se utiliza la variable volatile para evitar que el compilador optimice el bucle.

Medición de la Caché de Instrucciones:

Al final del código original, se añade un nuevo bloque que mide el tiempo de ejecución de la función dummy_function para diferentes tamaños de caché y pasos (stride). Este bloque funciona de manera similar al bloque original que mide el rendimiento de la caché de datos.

Impresión de Resultados:

Los resultados se imprimen en formato CSV, separando los resultados de la caché de datos y la caché de instrucciones con etiquetas claras.

Importancia de Conocer las Características de la Caché de Instrucciones

Conocer las características de la caché de instrucciones es crucial por varias razones:

- Conocer las características de la caché de instrucciones permite a los desarrolladores escribir código que se ejecute más rápido al reducir los fallos de caché y mejorar la localización temporal y espacial de las instrucciones.
- Los problemas de rendimiento a menudo se deben a fallos en la caché. Medir las características de la caché de instrucciones puede ayudar a identificar y corregir estos problemas.
- La información sobre el rendimiento de la caché de instrucciones es crucial para los diseñadores de hardware y sistemas operativos que buscan mejorar la eficiencia y el rendimiento de los sistemas informáticos.
- Conocer estas características permite comparar diferentes arquitecturas y configuraciones de hardware para tomar decisiones informadas sobre la compra y el uso de equipos.

Con estas modificaciones y la comprensión de su importancia, se pueden realizar pruebas más exhaustivas de las cachés de datos e instrucciones, proporcionando una visión más completa del rendimiento de un sistema.

III. Conclusiones

A. *Impacto del Tamaño de Caché y Stride en el Rendimiento:*

Los gráficos muestran cómo los tiempos de acceso a la memoria varían con diferentes tamaños de caché y diferentes strides.

Para el sistema con 16 GB de RAM DDR4 en Ryzen 3 3200g, los tiempos de acceso son consistentemente bajos hasta que los tamaños de caché alcanzan un umbral específico, tras el cual los tiempos aumentan significativamente.

En el sistema con 8 GB de RAM en AMD 3020e, se observa un comportamiento similar, pero con tiempos de acceso más elevados en general, reflejando la diferencia en la arquitectura y capacidad de los sistemas evaluados.

B. *Optimización del Hardware y Software:*

Los resultados del estudio permiten identificar las configuraciones de memoria que ofrecen el mejor rendimiento, ayudando a los diseñadores a tomar decisiones informadas sobre cómo estructurar los niveles de caché.

La capacidad de medir y comprender los tiempos de acceso a diferentes niveles de caché proporciona una base sólida para optimizar tanto el hardware como el software, logrando sistemas más eficientes y de mayor rendimiento.

En resumen, el estudio detallado del sistema de memoria mediante el programa presentado permite una comprensión profunda de cómo diferentes configuraciones afectan el rendimiento. Esto es crucial para el desarrollo de sistemas de alta eficiencia y rendimiento, y proporciona una valiosa herramienta educativa y profesional.

C. *Importancia en la relacion entre cache y rendimiento*

La comparación de resultados entre diferentes configuraciones de hardware resalta la importancia de conocer y ajustar las características de la caché de instrucciones. Para futuras mejoras y pruebas adicionales, se podrían considerar modificaciones en el código para evaluar más aspectos específicos de la caché, como latencia y eficiencia en diferentes patrones de acceso. Comprender estas características es esencial para ingenieros de sistemas y computación al diseñar y optimizar arquitecturas de computadores, asegurando un rendimiento robusto y eficiente.

En resumen, este proyecto ha proporcionado valiosos insights sobre el comportamiento de la memoria y las jerarquías de caché, subrayando la importancia de herramientas de evaluación precisas y el análisis detallado en la toma de decisiones informadas en ingeniería de

Referencias

[1] Barrios, C. J. (s/f). Computer Architecture A Quantitative Approach, Fifth Edition. http://wiki.sc3.uis.edu.co. Recuperado el 22 de junio de 2024, de http://wiki.sc3.uis.edu.co/images/d/d4/MemoryH.pdf. Pagina 4.

[2] Barrios, C. J. (s/f). Computer Architecture A Quantitative Approach, Fifth Edition. http://wiki.sc3.uis.edu.co. Recuperado el 22 de junio de 2024, de http://wiki.sc3.uis.edu.co/images/d/d4/MemoryH.pdf. Pagina 10.

[3] Casos de Estudio y Ejercicios por Norma P. Jouppi, Naveen Muralimanohar, y Sheng LI en el libro Patterson and Hennesy, Computer Architecture; A Quantitative Approach. Página 133 de la Quinta Edición.