

**ORGANIZACIÓN DE COMPUTADORES: LABORATORIO 3**  
**ANÁLISIS DE CACHÉ**

**NICOLÁS OLIVARES**

Profesores:

Felipe Garay

Erika Rosas

Nicolás Hidalgo

Ayudante:

Francisco López



# TABLA DE CONTENIDOS

<b>ÍNDICE DE FIGURAS .....</b>	<b>v</b>
<b>ÍNDICE DE CUADROS .....</b>	<b>vii</b>
<b>CAPÍTULO 1. INTRODUCCIÓN .....</b>	<b>9</b>
1.1 MOTIVACIÓN . . . . .	9
1.2 PROBLEMA . . . . .	9
1.3 OBJETIVOS . . . . .	9
1.3.1 Objetivos generales . . . . .	9
1.3.2 Objetivos específicos . . . . .	9
1.4 ORGANIZACIÓN DEL DOCUMENTO . . . . .	10
1.5 HERRAMIENTAS . . . . .	10
<b>CAPÍTULO 2. MARCO TEÓRICO .....</b>	<b>11</b>
2.1 CACHÉ . . . . .	11
2.1.1 Localidad: temporal y espacial . . . . .	11
2.1.2 Tasas: Hit y Miss . . . . .	12
2.2 ALGORITMOS . . . . .	12
2.2.1 Selection sort . . . . .	12
2.2.2 Merge sort . . . . .	13
2.3 LISTAS ENLAZADAS . . . . .	15
2.3.1 Tipos y características . . . . .	15
2.3.2 Listas enlazadas vs otras estructuras . . . . .	16
2.4 FUZZ TESTING . . . . .	16
<b>CAPÍTULO 3. DESARROLLO .....</b>	<b>17</b>
3.1 PRELIMINARES . . . . .	17
3.1.1 Elección de herramientas . . . . .	17
3.2 ALGORITMOS . . . . .	17
3.3 Implementación . . . . .	17
3.3.1 Lectura y escritura . . . . .	17
3.3.2 Distribución de directorio de trabajo . . . . .	18
Notación . . . . .	19

Ejecución . . . . .	19
3.3.3 Programa de pruebas . . . . .	19
3.3.4 TDA de las listas enlazadas . . . . .	21
3.3.5 <i>Selection sort</i> con listas enlazadas . . . . .	21
3.3.6 <i>Merge sort</i> con listas enlazadas . . . . .	21
3.3.7 <i>Merge sort</i> con arreglos de memoria contigua . . . . .	22
3.3.8 <i>Merge sort</i> con arreglos de memoria contigua (Hipótesis) . . . . .	22
3.4 RESULTADOS . . . . .	22
<b>CAPÍTULO 4. ANÁLISIS.....</b>	<b>29</b>
4.1 COMPARACIÓN DE RENDIMIENTO DE IMPLEMENTACIÓN CON LISTAS EN- LAZADAS . . . . .	29
4.1.1 Observaciones previas . . . . .	29
4.1.2 Análisis de observaciones . . . . .	29
4.1.3 Selection vs Merge . . . . .	30
Elección . . . . .	30
4.2 EVALUACIÓN DE IMPLEMENTACIÓN CON ARREGLOS . . . . .	31
4.2.1 Observaciones previas . . . . .	31
4.2.2 Análisis de observaciones . . . . .	31
<b>CAPÍTULO 5. CONCLUSIÓN .....</b>	<b>33</b>
5.1 OBJETIVOS PLANTEADOS . . . . .	33
5.2 ALCANCES DE RESULTADOS . . . . .	33
5.3 Expectativas . . . . .	33
<b>CAPÍTULO 6. ANEXO.....</b>	<b>35</b>
6.1 RESULTADOS DE CONFIGURACIONES DE CACHE . . . . .	35
<b>CAPÍTULO 7. BIBLIOGRAFÍA .....</b>	<b>65</b>

# ÍNDICE DE FIGURAS

Figura 2-1: Evolución de la mejora de CPU y memoria DRAM [4] . . . . .	11
Figura 2-2: Esquema de un nodo y una lista enlazada [9] . . . . .	15
Figura 2-3: Lista doblemente enlazada[10] . . . . .	15
Figura 2-4: Lista circular [11] . . . . .	16
Figura 4-1: Resultados test de Wilcoxon . . . . .	30
Figura 4-2: Resultados test de Wilcoxon para hipótesis planteada 4.3 . . . . .	32
Figura 6-1: Test de caché: Selección ID = 01 . . . . .	35
Figura 6-2: Test de caché: Selección ID = 02 . . . . .	36
Figura 6-3: Test de caché: Selección ID = 03 . . . . .	37
Figura 6-4: Test de caché: Selección ID = 04 . . . . .	38
Figura 6-5: Test de caché: Selección ID = 05 . . . . .	39
Figura 6-6: Test de caché: Selección ID = 06 . . . . .	40
Figura 6-7: Test de caché: Selección ID = 07 . . . . .	41
Figura 6-8: Test de caché: Selección ID = 08 . . . . .	42
Figura 6-9: Test de caché: Selección ID = 09 . . . . .	43
Figura 6-10: Test de caché: Mezcla ID = 01 . . . . .	44
Figura 6-11: Test de caché: Mezcla ID = 02 . . . . .	45
Figura 6-12: Test de caché: Mezcla ID = 03 . . . . .	46
Figura 6-13: Test de caché: Mezcla ID = 04 . . . . .	47
Figura 6-14: Test de caché: Mezcla ID = 05 . . . . .	48
Figura 6-15: Test de caché: Mezcla ID = 06 . . . . .	49
Figura 6-16: Test de caché: Mezcla ID = 07 . . . . .	50
Figura 6-17: Test de caché: Mezcla ID = 08 . . . . .	51
Figura 6-18: Test de caché: Mezcla ID = 09 . . . . .	52
Figura 6-19: Test de caché: Programa con arreglos ID = 01 . . . . .	53
Figura 6-20: Test de caché: Programa con arreglos ID = 02 . . . . .	54
Figura 6-21: Test de caché: Programa con arreglos ID = 03 . . . . .	55
Figura 6-22: Test de caché: Programa con arreglos ID = 04 . . . . .	56
Figura 6-23: Test de caché: Programa con arreglos ID = 05 . . . . .	57
Figura 6-24: Test de caché: Programa con arreglos ID = 06 . . . . .	58
Figura 6-25: Test de caché: Programa con arreglos con mejora ID = 01 . . . . .	59
Figura 6-26: Test de caché: Programa con arreglos con mejora ID = 02 . . . . .	60

---

Figura 6-27: Test de caché: Programa con arreglos con mejora ID = 03 . . . . .	61
Figura 6-28: Test de caché: Programa con arreglos con mejora ID = 04 . . . . .	62
Figura 6-29: Test de caché: Programa con arreglos con mejora ID = 05 . . . . .	63
Figura 6-30: Test de caché: Programa con arreglos con mejora ID = 06 . . . . .	64

# ÍNDICE DE CUADROS

Tabla 3.1: Resultados de caché, algoritmo selección sobre listas enlazadas . . . . .	24
Tabla 3.2: Resultados de caché, algoritmo mezcla para listas enlazadas . . . . .	25
Tabla 3.3: Resultados de caché, algoritmo mezcla para arreglos estáticos . . . . .	26
Tabla 3.4: Resultados de caché, algoritmo mezcla para arreglos estáticos (Hipótesis) . .	27





# **CAPÍTULO 1. INTRODUCCIÓN**

Se presentan a continuación los aspectos del preámbulo de este documento.

## **1.1 MOTIVACIÓN**

Sea de conocimiento al lector, que el presente informe corresponde a un reporte analítico del desarrollo de la experiencia número 3 del laboratorio correspondiente a la asignatura de Organización de Computadores impartida por el Departamento de Informática de la Universidad de Santiago de Chile. Este laboratorio busca la expansión del conocimiento del lector, y su compromiso en la búsqueda de mejores combinaciones de tecnologías a las que se puede recurrir en la construcción de elementos computacionales eficientes para el uso de las personas.

## **1.2 PROBLEMA**

Se presenta el enunciado de la experiencia 3 del laboratorio de la asignatura [1]. Se busca obtener programas computacionales basados en algoritmos de ordenamiento que sean sometidos a una serie de pruebas y configuraciones de caché de memoria.

Los descritos programas han de ser escritos en lenguaje de bajo nivel ensamblador MIPS [2]. Se tiene, además, el deseo de construir un programa de pruebas en un lenguaje de alto nivel que verifique la integridad de los programas de ordenamiento.

## **1.3 OBJETIVOS**

### **1.3.1 Objetivos generales**

Establecer, con fundamentos analíticos, la relación entre el uso de diversas configuraciones de caché y el uso de distintas técnicas de algoritmos e implementación de estos.

### **1.3.2 Objetivos específicos**

Se definen para este laboratorio los siguientes objetivos específicos [1]:

- Construir programa de pruebas, con la estrategia de Fuzzy Testing para comprobar el correcto funcionamiento de los programas que se mencionan posteriormente.
- Construir programa en lenguaje ensamblador MIPS, con un algoritmo iterativo de ordenamiento aplicado a listas enlazadas.
- Construir programa en lenguaje ensamblador MIPS, con un algoritmo recursivo de ordenamiento aplicado a listas enlazadas.
- Comparar programas hechos con listas enlazadas, elegir aquel con peor desempeño e implementarlo con arreglos.

- Plantear una hipótesis sobre el programa implementado con arreglos.
- Análisis de resultados obtenidos.

## 1.4 ORGANIZACIÓN DEL DOCUMENTO

En esta sección el lector podrá dimensionar la estructura y distribución de la información de este documento. Este informe consta de las siguientes partes:

- Introducción: Preámbulo del documento, con detalle y definición de objetivos de la experiencia.
- Marco Teórico: Nivelación de conocimientos al lector, para reconocer los conceptos tratados en capítulos posteriores.
- Desarrollo: Se aborda la realización detallada de la experiencia, para entender el desarrollo de los programas pedidos y así también el razonamiento puesto en ellos.
- Análisis: Fase de aplicación de conceptos de experimentación y análisis de las soluciones construidas.
- Conclusión: Evaluación comparativa de objetivos y reacciones finales al proceso de la experiencia.

## 1.5 HERRAMIENTAS

Las herramientas utilizadas en este laboratorio son las siguientes

- Sistema Operativo Linux Mint 18
- Git para control de versiones y seguimiento del trabajo. Repositorio en GitHub. [3]
- Editor de texto Atom v1.11.2
- Lenguaje de programación Python v2.7.3
- Simulador de MIPS MARS v4.5 (con modificaciones hechas por profesor Felipe Garay)
- Rkward software de R para Linux v4.14.16

## CAPÍTULO 2. MARCO TEÓRICO

### 2.1 CACHÉ

La velocidad de la memoria se ha distanciado progresivamente de la velocidad de los procesadores. En la figura siguiente se muestran las gráficas de la evolución experimentada por el rendimiento de las CPUs y las memoria DRAM (soporte de la memoria principal de los computadores actuales) en los últimos años. Las curvas muestran que el rendimiento de la CPU aumentó un 35% anual desde 1980 hasta 1987; y un 55% anual a partir de ese año. En cambio la memoria ha mantenido un crecimiento sostenido del 7% anual desde 1980 hasta la fecha. Esto significa que si se mantiene la tendencia, el diferencial de rendimiento no sólo se mantendrá sino que aumentará en el futuro. Para equilibrar esta diferencia se viene utilizando una solución arquitectónica: la memoria caché

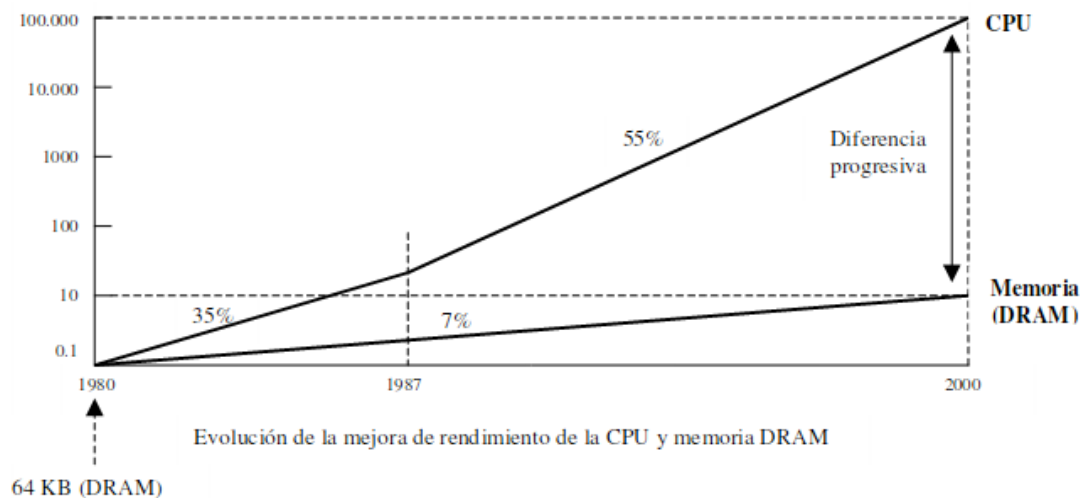


Figura 2-1: Evolución de la mejora de CPU y memoria DRAM [4]

La memoria caché es una memoria pequeña y rápida que se interpone entre la CPU y la memoria principal para que el conjunto opere a mayor velocidad. Para ello es necesario mantener en la caché aquellas zonas de la memoria principal con mayor probabilidad de ser referenciadas. Esto es posible gracias a la propiedad de localidad de referencia de los programas.

#### 2.1.1 Localidad: temporal y espacial

Los programas manifiestan una propiedad que se explota en el diseño del sistema de gestión de memoria de los computadores en general y de la memoria caché en particular, la localidad de referencias: los programas tienden a reutilizar los datos e instrucciones que utilizaron recientemente. Una regla empírica que se suele cumplir en la mayoría de los programas revela que gastan el 90% de su tiempo de ejecución sobre sólo el 10% de su código. Una consecuencia de

la localidad de referencia es que se puede predecir con razonable precisión las instrucciones y datos que el programa utilizará en el futuro cercano a partir del conocimiento de los accesos a memoria realizados en el pasado reciente. La localidad de referencia se manifiesta en una doble dimensión: temporal y espacial.

**Localidad temporal:** las palabras de memoria accedidas recientemente tienen una alta probabilidad de volver a ser accedidas en el futuro cercano. La localidad temporal de los programas viene motivada principalmente por la existencia de bucles.

**Localidad espacial:** las palabras próximas en el espacio de memoria a las recientemente referenciadas tienen una alta probabilidad de ser también referenciadas en el futuro cercano. Es decir, que las palabras próximas en memoria tienden a ser referenciadas juntas en el tiempo. La localidad espacial viene motivada fundamentalmente por la linealidad de los programas (secuenciamiento lineal de las instrucciones) y el acceso a las estructuras de datos regulares.[4]

### 2.1.2 Tasas: Hit y Miss

Cuando una dirección se presenta en el sistema caché pueden ocurrir dos cosas [4]:

Acierto de caché (*hit*): el contenido de la dirección se encuentre en un bloque ubicado en una línea de la caché.

Fallo de caché (*miss*): el contenido de la dirección no se encuentre en ningún bloque ubicado en alguna línea de la caché.

Si en la ejecución de un programa se realizan  $Nr$  referencias a memoria, de las que  $Na$  son aciertos caché y  $Nf$  fallos caché, se definen los siguientes valores:

- Tasa de aciertos:  $Ta = Na/Nr$
- Tasa de fallos:  $Tf = Nf/Nr$

Evidentemente se cumple:

$$Ta = 1 - Tf$$

## 2.2 ALGORITMOS

La complejidad de un algoritmo responde a diferentes factores, de los cuales se destacan [5]:

- Complejidad temporal: cuanto se demora un algoritmo en terminar.
- Complejidad espacial: cuanta memoria operativa (RAM usualmente) es requerida por el algoritmo. Esto tiene dos apartados, la cantidad de memoria que necesita el código y la cantidad que necesitan los datos sobre los que opera el algoritmo.

### 2.2.1 Selection sort

Este algoritmo *Selection sort* corresponde a un algoritmo de carácter iterativo, con una complejidad promedio temporal de  $O(n^2)$  donde  $n$  es la cantidad de números a ordenar [6] y una complejidad espacial de también  $O(n^2)$ . Esta complejidad es obtenida bajo la operación del algoritmo sobre arreglos. Ver Algoritmo 1

**Algorithm 1** Ordenamiento por selección o *selection sort* [6]

---

```

1: procedure seleccion(lista)
2:   for  $i = 1$  hasta  $n - 1$  do
3:      $minimo = i$ 
4:     for  $j = i + 1$  hasta  $n$  do
5:       if  $lista[j] < lista[minimo]$  then
6:          $mínimo = j$ 
7:       end if
8:     end for
9:     intercambiar( $lista[i]$ ,  $lista[mínimo]$ )
10:  end for
11: end procedure

```

---

**2.2.2 Merge sort**

Este es un algoritmo de carácter recursivo que aplica la técnica de divide y vencerás. Es reconocido como uno de los algoritmos con mejor desempeño, por considerarse estable[7]. Su complejidad temporal corresponde en el peor caso es de  $O(n \log n)$ [7] y su complejidad espacial corresponde a  $O(n \log n)$  al igual que Quicksort.

El ordenamiento por mezcla incorpora dos ideas principales para mejorar su tiempo de ejecución [7]:

- Una lista pequeña necesitará menos pasos para ordenarse que una lista grande.
- Se necesitan menos pasos para construir una lista ordenada a partir de dos listas también ordenadas, que a partir de dos listas desordenadas. Por ejemplo, sólo será necesario entrelazar cada lista una vez que están ordenadas.

En el Algoritmo 2 se puede ver el uso de la recursión y la división realizada por el algoritmo de mezcla

**Algorithm 2** Ordenamiento por mezcla o *merge sort* [8]

---

```

1: procedure mezcla(a, inicio, fin)
2:   if  $inicio < fin$  then
3:      $medio = \frac{inicio + fin}{2}$ 
4:     mezcla(a, inicio, medio)
5:     mezcla(a, medio + 1, fin)
6:     combinar(a, inicio, medio, fin)
7:   end if
8: end procedure

```

---

El trabajo realizado por el algoritmo, descompone la lista a hasta los casos bases donde se tienen arreglos de un elemento, es decir, ya ordenados y procede a *combinar* estas soluciones. Ver Algoritmo 3

---

**Algorithm 3** Combinación en ordenamiento por mezcla o *merge sort* [8]
 

---

```

1: procedure combinar(a, inicio, medio, fin)
2:   Arreglo aux[fin - inicio + 1]
3:   h = inicio
4:   i = inicio
5:   j = medio + 1
6:   while (i ≤ medio) and (j ≤ fin) do
7:     if a[i] ≤ a[j] then
8:       aux[h] = a[i]
9:       i = i + 1
10:    else
11:      aux[h] = a[j]
12:      j = j + 1
13:    end if
14:    h = h + 1
15:  end while
16:  if i > medio then
17:    for k=j hasta fin do
18:      aux[h] = a[k]
19:      h = h + 1
20:    end for
21:  else
22:    for k=i hasta medio do
23:      aux[h] = a[k]
24:      h = h + 1
25:    end for
26:  end if
27:  for k=inicio hasta fin do
28:    a[k] = aux[k]
29:  end for
30: end procedure

```

---

## 2.3 LISTAS ENLAZADAS

La lista enlazada es un TDA que nos permite almacenar datos de una forma organizada, al igual que los vectores pero, a diferencia de estos, esta estructura es dinámica, por lo que no tenemos que saber "a priori" los elementos que puede contener.[9]

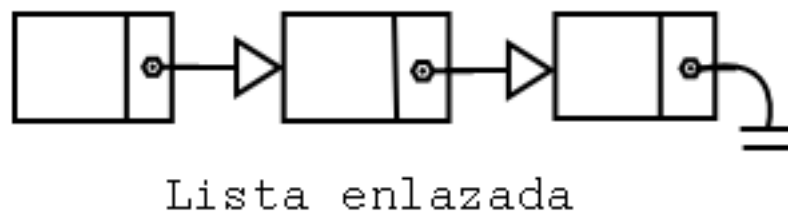
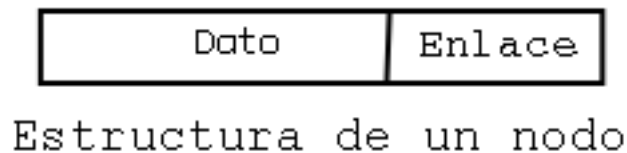


Figura 2-2: Esquema de un nodo y una lista enlazada [9]

### 2.3.1 Tipos y características

Existen distintos tipos de listas enlazadas, de los cuales se tiene: *Simples*, en este tipo, cada nodo presenta una referencia al siguiente nodo en la lista, y el último nodo de la lista referencia al valor nulo, ver Figura 2-2.

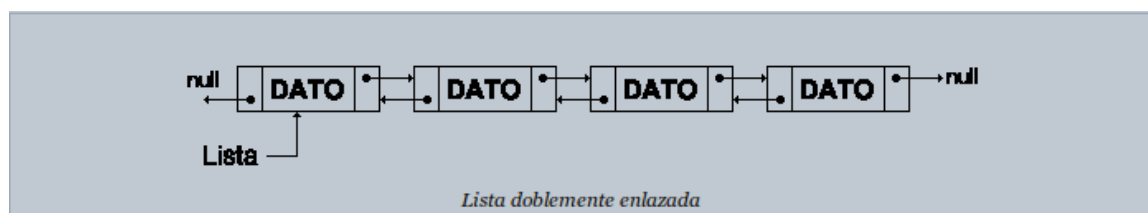


Figura 2-3: Lista doblemente enlazada[10]

También las hay *Dobles* (Figura 2-3), corresponde a un concepto similar a las listas enlazadas simples pero cada nodo consta con un puntero extra para referenciar al elemento anterior la lista y para el primer y último se tienen terminaciones al valor nulo.

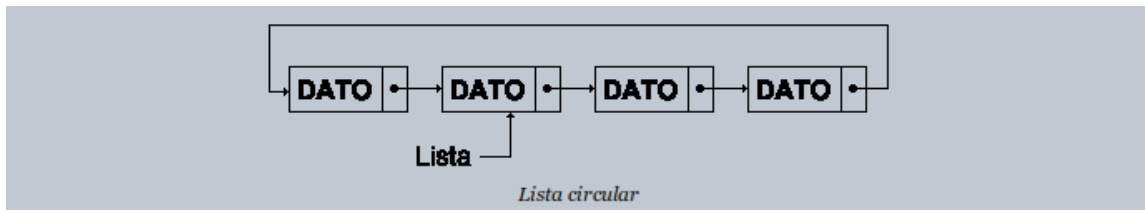


Figura 2-4: Lista circular [11]

Finalmente se tiene las listas enlazadas *Circulares* (Figura 2-4), estas pueden ser implementadas como *Simples* o *Dobles*, lo que las diferencia corresponde a su referencia en los nodos inicial y final ya que estos están conectados entre sí.

### 2.3.2 Listas enlazadas vs otras estructuras

Existen otros tipos de estructuras destinadas al almacenamiento de elementos de manera lineal unidimensional, como es el caso de los *Array* y la clase *Vector* [12][13]. Cada una de ellas tiene ventajas y desventajas comparativas con respecto a las listas enlazadas.

Una de las ventajas de usar la estructura *Vector* o *Array* es principalmente que las direcciones de memoria de los elementos en una estructura de este tipo son contiguas y sus valores son indexados de manera que el orden de complejidad de una búsqueda en este tipo de estructura es  $O(1)$  constante. A diferencia de las listas enlazadas, como se expresa anteriormente [9], las cuales poseen una capacidad de organizar sus elementos a lo largo de la memoria sin necesidad de estar en posiciones contiguas de memoria, lo que ocasiona que se dificulte el uso de indexación de valores, o bien aumente la complejidad de estas operaciones.

## 2.4 FUZZ TESTING

Fuzz testing o fuzzing es una técnica de prueba de software, comúnmente automatizada o semi-automatizada, que implica proveer datos inválidos, inesperados o aleatorios a la entrada de un programa. (Traducido [14]).

Esta técnica es utilizada para observar excepciones dentro del código del programa y que no son previstas por el programador durante la implementación de éste.



## CAPÍTULO 3. DESARROLLO

En este capítulo se explicará en detalle la generación de la solución al problema planteado, desde la elección de herramientas hasta la obtención de resultados.

### 3.1 PRELIMINARES

#### 3.1.1 Elección de herramientas

Para empezar, se explica la elección de las herramientas utilizadas. Se solicita el uso de Linux para la realización de la experiencia debido a la calidad de exigencia en el enunciado.[1] Se integra el uso de Git para un desarrollo controlado de la solución, además del uso del editor de texto Atom para escribir el programa de pruebas.

La elección de Python como lenguaje de programación se realiza por una cuestión de simpleza en la programación al pertenecer a una clase de lenguajes de alto nivel. Y ya como recurso necesario para la realización y obtención de resultados, se utiliza el simulador MARS [2]

### 3.2 ALGORITMOS

Es justo recordar que durante la primera parte de este curso, se realizó la experiencia número uno de este laboratorio que enfocaba su resolución al rendimiento de procesadores, en el cuál se implementaron dos algoritmos, uno recursivo y otro iterativo, como fue el caso de *Quick sort* y *Bubble sort*, respectivamente. Es por lo anterior, que en esta ocasión se pide explícitamente la utilización de algoritmos diferentes a los anteriormente mencionados para efectuar su implementación.

Para esta experiencia se opta por el algoritmo iterativo *Selection sort* y en el caso del algoritmo recursivo, se elige *Merge sort*. Los algoritmos seguidos para la implementación están disponibles en el Marco Teórico de este documento, con sus respectivas referencias[6][7][8].

### 3.3 Implementación

En esta sección se reconocen todos los aspectos y detalles considerados en la creación de cada programa y etapa de la experiencia a nivel práctico.

#### 3.3.1 Lectura y escritura

Para la lectura y escritura de archivos en MARS, se realiza un proceso de descomposición de carácter por carácter para efectuar la lectura de los valores del archivo, para efectos de la generación de cada número entero, se lee el archivo desde el final hacia el inicio, reconociendo el carácter de dígito, salto de línea o signos negativos, para identificar el momento en que se deben ingresar datos generados a la lista enlazada o estructura utilizada según corresponda. Para esta

lectura se usa un de bytes lo suficientemente grande para alcanzar un límite considerable de números para la lectura, considerando que el *INT\_MAX* tiene 10 dígitos, más el signo negativo y el salto de línea hacen un total de 12 caracteres máximos por línea, por lo tanto dado el buffer de  $2^{20}$  se aproxima una suficiencia para 87000 números por archivo.

Para la escritura se realiza el proceso inverso de conversión de entero a string para ser insertado en un archivo, en base a la descomposición de la división por la base 10 y concatenando el resto al buffer de escritura.

### 3.3.2 Distribución de directorio de trabajo

En la carpeta de código fuente de este laboratorio llamada *src/* usted encuentra la siguiente lista de carpetas y archivos:

- *src/fuzzy.py*: Corresponde al programa de pruebas. Para ejecutar usar el siguiente comando en la terminal:

```
>python fuzzy.py params
```

donde *params* corresponden a los parámetros del programa separados por un espacio cada uno:

1. Numero de pruebas aleatorias a generar
2. Valor mínimo del rango de datos a ordenar
3. Valor máximo del rango de datos a ordenar
4. Número de datos a ordenar

Los siguientes archivos pueden ser ejecutados sólo en MARS para obtener los resultados de las pruebas de cache con el Data Cache Simulator, debido a que reciben una entrada fija, por lo tanto no funcionan desde la línea de comandos.

- *src/mezcla.asm*
- *src/mezcla\_arreglos.asm*
- *src/seleccion.asm*

Ahora bien, los archivos que poseen el sufijo *testing*, corresponden a los que se ejecutan desde el programa de pruebas *fuzzy.py*, en este caso, estos programas funcionan al ejecutarlos desde la línea de comando por separado.

- *src/mezcla\_testing.asm*
- *src/mezcla\_arreglos\_testing.asm*
- *src/seleccion\_testing.asm*

Para ejecutarlos por separado y probar los programas de ordenamiento usar este comando:  
>*java -jar Mars\_cache.jar pa params*  
donde *params* corresponden a los parámetros del programa separados por un espacio cada uno:

1. Nombre archivo de entrada
2. Nombre archivo de salida

Dentro de los directorios extras, se encuentran:

- *src/pruebas\_cache*: En este directorio se encuentran las entradas fijas de los programas sobre los cuales se prueban las configuraciones de cache. También este directorio almacena la salida de estos programas.

Para el programa de pruebas se autogeneran dos directorios nuevos.

- *src/input*: en este directorio se generan los casos de prueba aleatorios durante la ejecución de *fuzzy.py*
- *src/output*: en este directorio se generan las salidas de los programas durante la ejecución de *fuzzy.py*

### **Notación**

En el caso de los archivos autogenerados en el programa de pruebas se tiene la siguiente notación:

- Archivos de entrada en *src/input*: CP\_{NOMBRE\_PROGRAMA}\_{IDENTIFICADOR}.txt
- Archivos de salida en *src/output*: OP\_{NOMBRE\_PROGRAMA}\_{IDENTIFICADOR}.txt

Existe un caso especial de notación, dentro del programa de pruebas se generan salidas intermedias que corresponden al testing de la prueba de *sort(sort)*, para los cuales se tiene la notación:

- *src/output*: OP\_{NOMBRE\_PROGRAMA}\_orden\_orden\_{IDENTIFICADOR}.txt

### **Ejecución**

El código fuente de esta experiencia cuenta con un archivo *makefile* que generará una ejecución única del programa de pruebas con un conjunto de 100 pruebas a una entrada de 50 elementos con rango entre [-2147483647,2147483647]. Por lo que si se desean realizar otra configuración de pruebas es necesario la ejecución independiente del programa mediante la terminal.

Para ejecutar el makefile: >make -f makefile

Para probar las pruebas realizadas al cache es necesario abrir el programa MARS y abrir los programas independientemente.

>java -jar Mars\_cache.jar

El programa Mars\_cache.jar está disponible en *src/*.

### **3.3.3 Programa de pruebas**

Para la construcción del programa de pruebas se usa el paradigma imperativo de programación, utilizando el lenguaje Python.

Se necesita construir un programa de pruebas que pudiera realizar un testing de los programas realizados en ensamblador MIPS y verificar que sus salidas son correctas en términos de obtener archivos de salida con conjuntos ordenados de menor a mayor. Por lo tanto, las pruebas realizadas sobre los archivos de salida buscan comprobar y chequear propiedades de conjuntos ordenados y de los algoritmos. Las propiedades evaluadas se clasifican en dos grupos, aquellas que responden a propiedades de la relación establecida para el conjunto ordenado, es decir, se sabe que un conjunto ordenado le corresponde un subconjunto numérico, en este caso de enteros  $A \subset \mathbb{Z}$ , y una relación binaria de orden, en este caso *menor o igual* o  $\leq$ . Y el otro grupo de propiedades corresponde a las propiedades propias de una función de ordenamiento.[15]

Propiedades destinadas a evaluación en el programa:

- **Reflexividad:**  $a \leq a$ . Esto proyectado a cada elemento del conjunto quiere decir que cada uno de los elementos es menor o igual que el mismo elemento. En el programa de pruebas esta propiedad comprueba para cada elemento del conjunto si alguno no cumple la propiedad, es decir, siendo  $L$  el conjunto, dado un  $a \in L$  se tiene que si  $a > a$  la propiedad no se cumpliría.
- **Transitividad:**  $a \leq b \wedge b \leq c \Rightarrow a \leq c$ . Esto se prueba en el programa de pruebas, seleccionando un número aleatorio  $a \in L$  y se toma el siguiente elemento  $b$  tal que  $a \leq b$  y luego se toma un tercer elemento  $c$  tal que  $b \leq c$  y se verifica que se cumpla que  $a \leq c$ , si no se cumple, existe un error en el conjunto al no estar ordenado.
- **Antisimetría:**  $a \leq b \wedge b \leq a \Rightarrow a = b$ . Esto se pone a prueba en el programa, para cada par de números de la lista se comprueba la propiedad y se cuenta las veces que se cumple, ese número de veces debe ser igual o mayor al número de elementos en el conjunto, en caso de haber elementos repetidos.
- **Primer elemento:** Sea  $m$  el primer elemento del conjunto entonces  $m$  es menor o igual que todos los demás valores del conjunto. Por lo tanto se comprueba tomando el primer elemento de la lista y se compara con los elementos restantes, si alguno no se cumple, entonces la propiedad no se cumple.
- **Último elemento:** Análoga a la anterior. Se toma el último valor o elemento y se compara con el resto de elementos del conjunto, si este es menor que algún otro valor, entonces la propiedad no se cumple.
- **Secuencia:** Se cumple cuando los elementos del conjunto son puestos como una secuencia  $\dots, e_i, e_{i+1}, \dots$  entonces para todos los elementos del conjunto se cumple que  $e_i \leq e_{i+1}$
- **Invertir dos veces una lista:** Esta propiedad se comprueba sobre el conjunto y si el conjunto sigue siendo el mismo luego de ser invertido dos veces, entonces la propiedad se cumple.
- **Ordenar 2 veces:** Esta propiedad se cumple si al aplicar el algoritmo de ordenamiento correspondiente dos veces  $sort(sort(L)) = sort(L)$ , este conjunto sigue siendo el mismo.

En el programa esta prueba se ejecuta luego de ya haber ordenado una vez el archivo de entrada correspondiente y luego al aplicar un nuevo ordenamiento en el programa ensamblador, se comprueba que ambas listas son idénticas, si se cumple, la prueba se acepta como aprobada.

### 3.3.4 TDA de las listas enlazadas

Se utilizan listas enlazadas simples para la implementación del laboratorio, ver página 15. Para el manejo de las listas enlazadas se implementa un TDA con las siguientes funciones:

- **es\_vacia**: Evalúa si una lista está vacía
- **insertar\_inicio**: Agrega un nuevo elemento al inicio de la lista
- **insertar\_final**: Agrega un nuevo elemento al final de la lista
- **crear\_nodo**: Reserva memoria para un nodo con un dato en él.

Además se crean funciones de conversión como

- **integer\_to\_string**: Divide por 10 sucesivamente y concatena el resto en un buffer, considerando salto de línea y signo.
- **char\_to\_int**: Transforma el valor de un carácter ASCII a un valor de dígito entero entre 0 y 9.
- **int\_to\_char**: Inversa a la anterior

La descripción detallada de estos procedimientos así como sus entradas y salidas puede encontrarse en la documentación del código fuente disponible en conjunto a este documento.

### 3.3.5 Selection sort con listas enlazadas

Para este algoritmo se utiliza la recomendación entregada en el enunciado de la experiencia [1], en cuanto a la reserva de memoria para los nodos de la lista, con lo que cada nodo queda declarado en memoria como un par de bloques contiguos de 4 bytes cada uno, 8 bytes en total con el uso de syscall 9.

Los primeros 4 bytes sirven para almacenar el dato entero del nodo, cuyo comportamiento y rango está definido por las limitaciones de la entrada definida en el enunciado [1] es de el rango total de enteros  $[-INT\_MAX, INT\_MAX]$  y en los últimos 4 bytes se almacena una dirección de memoria al nodo siguiente de la lista.

Para la implementación del algoritmo se realiza tal como el algoritmo descrito se plantea, realizando recorridos sobre la lista enlazada en cuestión e intercambiando los valores al final del ciclo interno, cuando ya se encuentre el mínimo de los elementos restantes por ordenar. Para el intercambio se utiliza el stack con el registro \$sp para reservar el valor temporal.

### 3.3.6 Merge sort con listas enlazadas

Al igual que el anterior con el uso de memoria, en este programa es idéntico. En este algoritmo planteado para merge sort y referenciado anteriormente, se requiere acceso a posiciones específicas durante el proceso de ordenamiento, para esto se agregan las siguientes funciones al TDA de lista enlazada:

- **insertar\_posicion:** inserta un elemento en una posición determinada de una lista enlazada.
- **dato\_en\_posicion:** obtiene el valor de un elemento en una posición determinada.

También se para el uso del arreglo auxiliar dentro del algoritmo se utiliza un arreglo global que es vaciado luego de completar una combinación, esta decisión se toma a partir de que se necesita tener acceso a posiciones variables para almacenar el orden temporal durante la combinación y luego este orden se traspasa a la lista original para reflejar el cambio al final del proceso.

### 3.3.7 Merge sort con arreglos de memoria contigua

Luego de haber mostrado en la fase de análisis cuál algoritmo de ordenamiento tenía un peor rendimiento comparativo entre ambos, siendo este el algoritmo de mezcla o merge sort, se procede a implementar con arreglos de memoria contigua.

Se intenta generar un TDA para trabajar con una estructura de datos como la que proporciona C++ con Vector [12][13], pero finalmente se opta por el uso de arreglos estáticos en memoria. Para obtener el tamaño o cantidad de datos se realiza una pre-lectura del archivo para contabilizar y luego una segunda lectura para obtener los datos en el arreglo.

El diseño del algoritmo usado está basado en arreglos [7][8] por lo que la implementación es intuitiva, pese a esto se modifica el uso del arreglo auxiliar, dejando a este fuera del procedimiento y no creándolo cada vez que se llama al procedimiento combinar.

Los resultados obtenidos están expresados en la Tabla 3-3

Ahora, diríjase a la página 31 para observar el análisis a estos resultados.

### 3.3.8 Merge sort con arreglos de memoria contigua (Hipótesis)

Planteada la hipótesis, se procede a construir la modificación al programa de ordenamiento de arreglos de memoria continua.

## 3.4 RESULTADOS

En esta sección se muestra el resumen de los resultados obtenidos para las pruebas realizadas sobre cada programa según la instancia correspondiente de la experiencia.

Para la obtención de los resultados de las distintas configuración de caché se aplica cada algoritmo a un conjunto de archivos de prueba generados de forma aleatoria con un máximo de 500 elementos enteros y con el rango completo de enteros, de tal manera que los tiempos de ejecución sean razonablemente, reproducibles y comprobables. Se aplican 18 experimentos para obtener

resultados, en cada uno se va cambiando la configuración del caché. En las siguientes tablas se encuentran los resultados obtenidos, detallando la organización del cache y sus correspondientes conteos de Hit, Miss y su Tasa de Hit.

Tabla 3.1: Resultados de caché, algoritmo selección sobre listas enlazadas

Test ID	Política de posicionamiento	Política de reemplazo de bloques	Nº de bloques	Tamaño de bloques (palabras)	Tamaño de conjunto (bloques)	Tamaño de caché (bytes)	Accesos a memoria	Hits	Misses	Tasa de Hit(%)
1	Direct Mapping	LRU	8	4	1	128	406203	208549	197654	51%
2	Fully Associative	LRU	8	4	8	128	406203	243324	162879	60%
3	N-way Set Associative	LRU	64	4	8	1024	406203	253812	152391	62%
4	Direct Mapping	LRU	64	16	1	4096	406203	276247	129956	68%
5	Fully Associative	LRU	128	16	128	8192	406203	283361	122842	70%
6	N-way Set Associative	LRU	128	64	16	32768	406203	302335	103868	74%
7	Direct Mapping	LRU	128	2048	1	1048576	406203	406167	36	100%
8	Fully Associative	LRU	128	2048	128	1048576	406203	406169	34	100%
9	N-way Set Associative	LRU	128	2048	16	1048576	406203	406168	35	100%
10	Direct Mapping	Random	8	4	1	128	406203	205258	200945	51%
11	Fully Associative	Random	8	4	8	128	406203	225482	180721	56%
12	N-way Set Associative	Random	64	4	8	1024	406203	245511	160692	60%
13	Direct Mapping	Random	64	16	1	4096	406203	272404	133799	67%
14	Fully Associative	Random	128	16	128	8192	406203	298907	107296	74%
15	N-way Set Associative	Random	128	64	16	32768	406203	317229	88974	78%
16	Direct Mapping	Random	128	2048	1	1048576	406203	406166	37	100%
17	Fully Associative	Random	128	2048	128	1048576	406203	406170	33	100%
18	N-way Set Associative	Random	128	2048	16	1048576	406203	406167	36	100%

Para mayor referencia de los resultados avalados con imágenes, ver Anexo en página 35.



Tabla 3.2: Resultados de caché, algoritmo mezcla para listas enlazadas

Test ID	Política de posicionamiento	Política de reemplazo de bloques	Nº de bloques	Tamaño de bloques (palabras)	Tamaño de conjunto (bloques)	Tamaño de caché (bytes)	Accesos a memoria	Hits	Misses	Tasa de Hit(%)
1	Direct Mapping	LRU	8	4	1	128	3310682	92766	3217916	3%
2	Fully Associative	LRU	8	4	8	128	3310682	81717	3228965	2%
3	N-way Set Associative	LRU	64	4	8	1024	3310682	170005	3140677	5%
4	Direct Mapping	LRU	64	16	1	4096	3310682	266219	3044463	8%
5	Fully Associative	LRU	128	16	128	8192	3310682	365832	2944850	11%
6	N-way Set Associative	LRU	128	64	16	32768	3310682	713550	2597123	22%
7	Direct Mapping	LRU	128	2048	1	1048576	3310682	3310644	38	100%
8	Fully Associative	LRU	128	2048	128	1048576	3310682	3310646	36	100%
9	N-way Set Associative	LRU	128	2048	16	1048576	3310682	3310647	35	100%
10	Direct Mapping	Random	8	4	1	128	3310682	92029	3218662	3%
11	Fully Associative	Random	8	4	8	128	3310682	84415	3226267	3%
12	N-way Set Associative	Random	64	4	8	1024	3310682	246804	3063878	7%
13	Direct Mapping	Random	64	16	1	4096	3310682	249187	3061495	8%
14	Fully Associative	Random	128	16	128	8192	3310682	815477	2495205	25%
15	N-way Set Associative	Random	128	64	16	32768	3310682	1033514	2277168	31%
16	Direct Mapping	Random	128	2048	1	1048576	3310682	6621322	42	100%
17	Fully Associative	Random	128	2048	128	1048576	3310682	6621328	36	100%
18	N-way Set Associative	Random	128	2048	16	1048576	3310682	6621329	35	100%

Para mayor referencia de los resultados avalados con imágenes, ver Anexo en página 35.

Tabla 3.3: Resultados de caché, algoritmo mezcla para arreglos estáticos

Test ID	Política de posicionamiento	Política de reemplazo de bloques	Nº de bloques	Tamaño de bloques (palabras)	Tamaño de conjunto (bloques)	Tamaño de caché (bytes)	Accesos a memoria	Hits	Misses	Tasa de Hit
1	Direct Mapping	LRU	8	4	1	128	61421	53335	8086	87%
2	Fully Associative	LRU	8	4	8	128	63361	60281	3080	95%
3	N-way Set Associative	LRU	64	4	8	1024	61677	60599	1078	98%
4	Direct Mapping	LRU	64	16	1	4096	64141	62138	2003	97%
5	Fully Associative	LRU	128	16	128	8192	60673	60598	75	100%
6	N-way Set Associative	LRU	128	64	16	32768	64141	64117	24	100%
7	Direct Mapping	Random	8	4	1	128	61147	53273	7874	87%
8	Fully Associative	Random	8	4	8	128	63383	59232	4151	93%
9	N-way Set Associative	Random	64	4	8	1024	61511	61433	78	100%
10	Direct Mapping	Random	64	16	1	4096	61329	61192	137	100%
11	Fully Associative	Random	128	16	128	8192	61325	61250	75	100%
12	N-way Set Associative	Random	128	64	16	32768	62481	62458	23	100%

Para mayor referencia de los resultados avalados con imágenes, ver Anexo en página 35.

Tabla 3.4: Resultados de caché, algoritmo mezcla para arreglos estáticos (Hipótesis)

Test ID	Política de posicionamiento	Política de reemplazo de bloques	Nº de bloques	Tamaño de bloques (palabras)	Tamaño de conjunto (bloques)	Tamaño de caché (bytes)	Accesos a memoria	Hits	Misses	Tasa de Hit
1	Direct Mapping	LRU	8	4	1	128	60257	51818	8439	86%
2	Fully Associative	LRU	8	4	8	128	60183	56987	3196	95%
3	N-way Set Associative	LRU	64	4	8	1024	60015	58350	1665	97%
4	Direct Mapping	LRU	64	16	1	4096	60181	59339	842	99%
5	Fully Associative	LRU	128	16	128	8192	60183	59734	449	99%
6	N-way Set Associative	LRU	128	64	16	32768	60183	59886	297	100%
7	Direct Mapping	Random	8	4	1	128	60249	51593	8656	86%
8	Fully Associative	Random	8	4	8	128	60257	55820	4437	93%
9	N-way Set Associative	Random	64	4	8	1024	60297	58336	1961	97%
10	Direct Mapping	Random	64	16	1	4096	60181	59548	633	99%
11	Fully Associative	Random	128	16	128	8192	59413	58941	472	99%
12	N-way Set Associative	Random	128	64	16	32768	60310	60004	297	100%

Para mayor referencia de los resultados avalados con imágenes, ver Anexo en página 35.



## **CAPÍTULO 4. ANÁLISIS**

### **4.1 COMPARACIÓN DE RENDIMIENTO DE IMPLEMENTACIÓN CON LISTAS ENLAZADAS**

#### **4.1.1 Observaciones previas**

Dentro del catastro de datos obtenidos se tiene un conjunto de observaciones que se presentan a continuación:

1. Se aplicaron experimentos con dos políticas de reemplazo diferente LRU y Random.
2. Se debe considerar el uso de accesos de memoria para la lectura y escritura de los archivos y las listas, dado que solo una parte de los accesos a memoria provienen del algoritmo en estudio.
3. Para ambos algoritmos, se observa que si se aumenta la capacidad del caché en cuanto al aumento de la cantidad de bloques, y por ende del tamaño total en bytes de este, entonces se visualiza un aumento la tasa de Hits del programa.

#### **4.1.2 Análisis de observaciones**

1. Las diferencias porcentuales entre el uso de LRU y Random en la misma configuración es a lo más de 4 puntos porcentuales favorables a Random, se puede decir que esto se produce debido a que para la política LRU es necesario mantener un monitoreo del bloque que corresponde ser borrado eventualmente, esto se traduce en necesidad de "hardware" y comunicación extra (considerando que es un simulador de caché). Mientras que Random es más simple de implementar considerando que no debe mantener un historial de los accesos.
2. En el programa de merge sort se puede observar un detalle que representa la observación hecha, cuando se inicia el programa la tasa de Hit se mantiene en un nivel alto cercano al 100%, pero a medida que llega a un punto alrededor de las 210000 lecturas de memoria, este valor de la tasa disminuye abruptamente a un rango entre los 2% a 8%. Esto hace presumir que en las lecturas previas a las 210000 no responden al algoritmo, sino más bien a la fase de lectura del archivo. Algo similar ocurre con selection sort, pero esta variación no se observa de manera drástica cómo su rival comparativo, de un 100% de tasa en la primera parte baja un rango entre los 50% y 78%.
3. Esta observación refleja lo expresado dentro de la teoría de esta materia [16], donde se asegura que el aumento de asociatividad, es decir, que al pasar de una configuración de Direct Mapping a las demás, se generan agrupaciones de bloques en el cache de manera

que se aprovecha el concepto de localidad espacial, produciendo una mayor probabilidad de encontrar los valores solicitados en cache.

Un alcance que se debe destacar es que el uso de configuraciones de cache muy grandes permiten tener una tasa pseudo perfecta de Hit con un 100%, esto debido a que los datos necesarios para realizar las operaciones son sostenidos en el cache en su totalidad durante la ejecución, no teniendo que recurrir a niveles más bajos de la jerarquía de memoria. Todo esto considerando que la entrada del programa para las pruebas de caché es de 500 elementos.

### 4.1.3 Selection vs Merge

A simple vista dados los resultados observados en la Tabla 3-1 y la Tabla 3-2 y considerando las tasas de hits de ambos para las configuraciones idénticas evaluadas, se tiene una especulación de que el algoritmo de merge es menos eficiente a nivel de acceso de memoria que el algoritmo de selection. Selection fue implementado usando listas enlazadas en su totalidad, mientras que para merge sort, si bien, la estructura principal que se modifica es la lista enlazada cargada del archivo, el uso de una arreglo estático para el arreglo auxiliar en la combinación, involucra que el algoritmo utilice este último para almacenar los valores ordenados parciales, y luego se vacíe dejándolo con valores 0 luego de la combinación para realizar la siguiente.

#### *Elección*

Para evaluar aquél algoritmo que se comporta de la peor manera, se considera realizar un test estadístico que establecerá la veracidad de nuestra hipótesis de que la proporción de hits del algoritmo de mezcla es menor que la proporción de hits del algoritmo de selección dados los datos obtenidos.

**Hipótesis:**

$$H_0 : p_m = p_s \quad (4.1)$$

$$H_A : p_m > p_s \quad (4.2)$$

Desarrollo usando lenguaje estadístico en lenguaje R se aplica el test no paramétrico de rangos de signos de Wilcoxon:

```
+ wilcox.test(x,y,alternative= "greater",paired = TRUE,correct =FALSE)
>>      Wilcoxon signed rank test

data:  x and y
V = 78, p-value = 0.001096
alternative hypothesis: true location shift is greater than 0
```

*Figura 4-1: Resultados test de Wilcoxon*

**Conclusión:**

En definitiva y en base a los resultados del test de rangos de signos de Wilcoxon realizado existe suficiente evidencia para rechazar la idea de que la proporción de hits entre ambos algoritmos es

igual y por lo tanto afirmamos con una confianza del 95% que la proporción de hits es mayor para el algoritmo de selección. Por lo tanto el peor algoritmo es merge sort.

Ahora revise la página 22 donde se explica el desarrollo del algoritmo con arreglos de memoria contigua.

## 4.2 EVALUACIÓN DE IMPLEMENTACIÓN CON ARREGLOS

Para la implementación de merge sort con arreglos, se realizó una simulación de caché con casi todas configuraciones aplicadas en la etapas con listas enlazadas pero sin considerar las configuraciones 7,8,9 y 16,17,18 debido a que su comportamiento cuasi impecable con altas tasas de hits, contrasta con su factibilidad, dado que el costo de implementar un cache de estos tamaños.([4] pág. 23). Se opta por las configuraciones de menor costo de implementación. Las pruebas fueron realizadas sobre archivos de 250 números, la mitad de la ejecución anterior.

### 4.2.1 Observaciones previas

Algunas de las observaciones rescatadas de los resultados se exponen a continuación:

1. A simple vista el rendimiento de merge sort, en relación a la tasa de hits es mejor para el algoritmo implementado con arreglos, que para el de listas enlazada.
2. Los accesos de memoria difieren entre cada configuración
3. La variación porcentual entre la aplicación de la política de reemplazo LRU o Random es de a lo más 3 puntos porcentuales

### 4.2.2 Análisis de observaciones

1. Esto se produce porque la implementación con arreglos involucra una mejora en el aspecto de localidad espacial, debido a que los bloques que buscan ser accedidos en la ejecución del programa se encuentran en memoria contigua.
2. Esta diferencia entre los accesos de memoria, se debe a un posible bug en el programa de ordenamiento implementado con arreglos estáticos.
3. Se sigue sosteniendo que las diferencias entre las políticas no es un factor que modifique la tasas de hit de manera excesiva, como si lo hace el uso de configuraciones diferentes.

Ahora se plante una hipótesis sobre el algoritmo implementado con arreglos.

Dado que como se a visto durante las fases de análisis y con los resultados, es probable que el hecho de usar un algoritmo generado afuera del algoritmo en si para hacer el orden, sea causante de la mejora de este algoritmo con respecto a los demás, basándose en la tasa de hits. Esto se explicaría ya que en el proceso de combinar las llamadas recursivas se debería obtener nuevas peticiones de memoria.

Dicho esto, se plantea mover la generación del arreglo dentro del procedimiento de de combinar.

**Hipótesis:** La proporción de hits del algoritmo de arreglos previo al cambio, es mayor a la proporción de hits que resulta de producirse el cambio.

$$H_0 : p_{sin\_cambio} = p_{con\_cambio} \quad (4.3)$$

$$H_A : p_{sin\_cambio} > p_{con\_cambio} \quad (4.4)$$

Aplicando el mismo test no paramétrico visto anteriormente, se tiene:

```
> wilcox.test(x,y,alternative="greater",paired = TRUE,correct =FALSE)
>>      Wilcoxon signed rank test

data:  x and y
V = 29, p-value = 0.05359
alternative hypothesis: true location shift is greater than 0
```

*Figura 4-2: Resultados test de Wilcoxon para hipótesis planteada 4.3*

**Conclusión:** En base a lo expuesto y los datos obtenidos, se puede afirmar con una confianza del 95% que la proporción de hits en el algoritmo con arreglos antes del cambio es mayor que la proporción de hits después del cambio. Esta conclusión está sujeta a una leve de probabilidad de error del tipo I, debido al p-value > 0.05, pero es una diferencia mínima.

Cabe destacar que el programa con el cambio tiene errores al ordenar, dejando sólo una parte de los números ordenados. Pese a esto, la cantidad de accesos a memoria que se realizan son los mismos que hiciera si estuviera ordenado la totalidad del conjunto, el único problema es que se almacenan se generan sw con ceros sobre la lista final en algún punto del algoritmo.



## CAPÍTULO 5. CONCLUSIÓN

### 5.1 OBJETIVOS PLANTEADOS

- En el aspecto concerniente a la construcción del programa de pruebas, se logró construir este de manera efectiva, permitiendo ejecutar el resto de los programas de ordenamiento con entradas aleatorias entregando resultados de la integridad de estas soluciones.
- Para los programas de ordenamiento que se basaron en el uso de listas enlazadas simples, se obtuvieron los resultados esperados. El ordenamiento es realizado con éxito, según las propiedades puestas a prueba en el programa de testing estos algoritmos entregan el resultados esperado y correcto.
- Por otro lado, los programas de ordenamiento implementados con arreglos fueron realizados, y se encuentran en un estado del 95% de satisfacción. Esto debido a que en algunos casos de prueba el programa de mezcla sin cambios entrega resultados fallidos para la prueba de  $sort(sort) = sort(L)$  y debido a que el segundo programa esta basado en el primero, también presenta errores. Pese a esto, en cuanto al análisis realizado, estos aspectos no influyen en los resultados, debido a que los accesos a memoria considerados, pese a que los datos fueran erróneos, de igual manera requieren el uso de la jerarquía de memoria para encontrar los datos.
- Finalmente, para el análisis, se aplicó un método de observación y explicación avalado en la teoría y con el uso de modelos estadísticos para la comprobación de resultados y eliminación de especulaciones.

### 5.2 ALCANCES DE RESULTADOS

En base a los resultados obtenidos, se puede destacar la adquisición de conocimientos y el fortalecimiento de aquellos conceptos vistos en la teoría de la asignatura. Se integraron conceptos de análisis, programación, que se aprecian como valiosos para la formación profesional, generando criterio y objetividad.

### 5.3 Expectativas

Luego de esta experiencia, se espera que el lector se intrigue con la búsqueda de información complementaria a este estudio, utilizando recursos de la literatura.

Está presente, de mi parte como autor de este documento y los programas adjuntos, el compromiso de profundizar los aspectos que no se abordaron totalmente durante la experiencia.



## CAPÍTULO 6. ANEXO

### 6.1 RESULTADOS DE CONFIGURACIONES DE CACHÉ

Estas figuras son presentadas para corroborar la realización de las pruebas, el detalle resumido puede ser visualizado en las tablas correspondientes en la presentación de Resultados

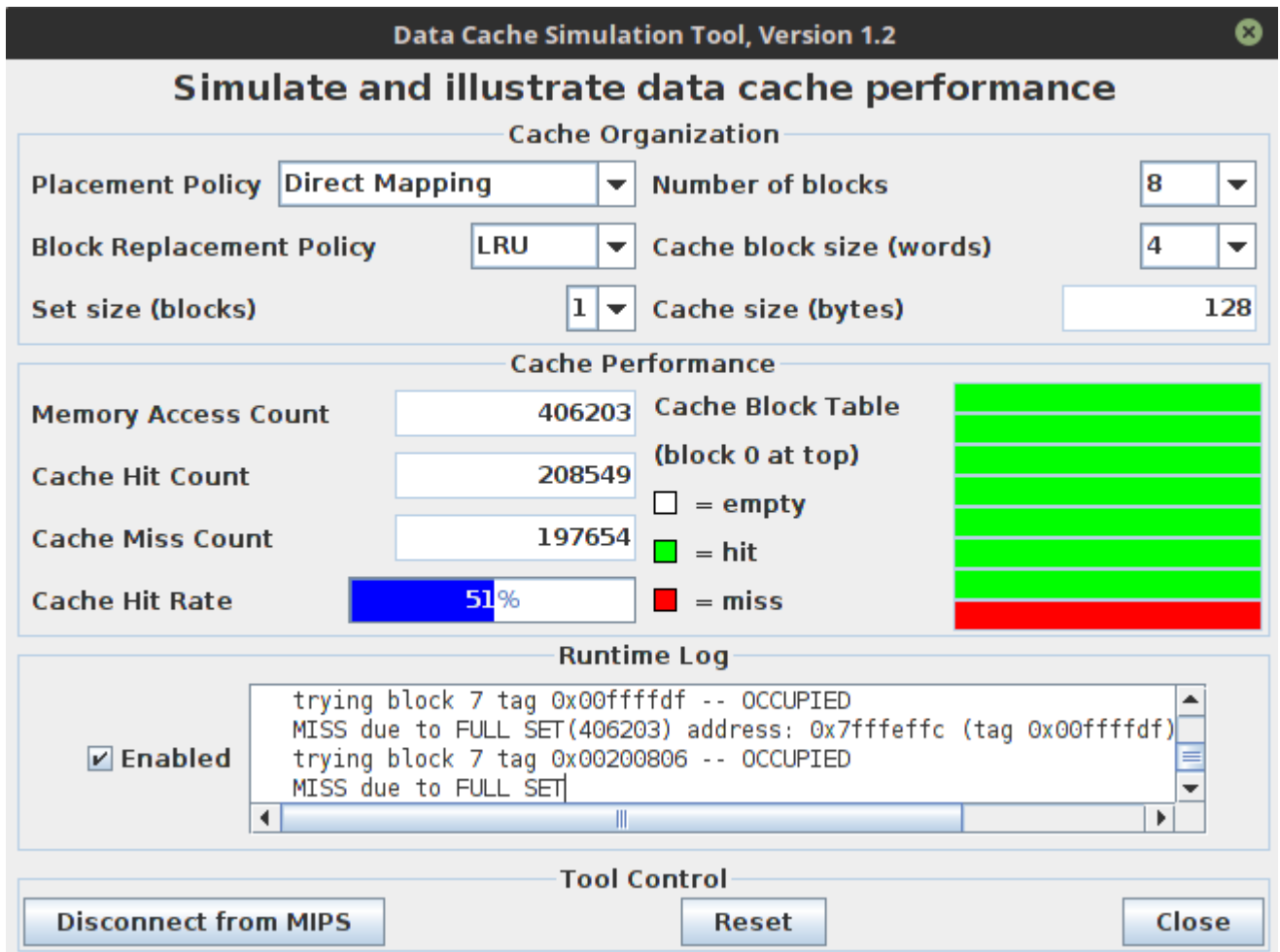


Figura 6-1: Test de caché: Selección ID = 01

**Data Cache Simulation Tool, Version 1.2**

---

**Simulate and illustrate data cache performance**

**Cache Organization**

Placement Policy	Fully Associative	Number of blocks	8
Block Replacement Policy	LRU	Cache block size (words)	4
Set size (blocks)	8	Cache size (bytes)	128

**Cache Performance**

Memory Access Count	406203	<b>Cache Block Table</b> (block 0 at top) <input type="checkbox"/> = empty <input checked="" type="checkbox"/> = hit <input checked="" type="checkbox"/> = miss	<div style="width: 100%; height: 10px; background-color: red;"></div>
Cache Hit Count	243324		<div style="width: 100%; height: 10px; background-color: green;"></div>
Cache Miss Count	162879		<div style="width: 100%; height: 10px; background-color: green;"></div>
Cache Hit Rate	60%		<div style="width: 100%; height: 10px; background-color: green;"></div>

**Runtime Log**

☐ Enabled

**Tool Control**

Disconnect from MIPSResetClose

Figura 6-2: Test de caché: Selección ID = 02

**Data Cache Simulation Tool, Version 1.2**

### Simulate and illustrate data cache performance

**Cache Organization**

Placement Policy: **N-way Set Associative** Number of blocks: **64**

Block Replacement Policy: **LRU** Cache block size (words): **4**

Set size (blocks): **8** Cache size (bytes): **1024**

**Cache Performance**

Memory Access Count: **406203**

Cache Hit Count: **253812**

Cache Miss Count: **152391**

Cache Hit Rate: **62%**

Cache Block Table (block 0 at top)

☐ = empty  
☒ = hit  
☐ = miss

**Runtime Log**

☐ Enabled

**Tool Control**

**Disconnect from MIPS** **Reset** **Close**

*Figura 6-3: Test de caché: Selección ID = 03*

**Data Cache Simulation Tool, Version 1.2**

**Simulate and illustrate data cache performance**

**Cache Organization**

Placement Policy: **Direct Mapping** Number of blocks: **64**

Block Replacement Policy: **LRU** Cache block size (words): **16**

Set size (blocks): **1** Cache size (bytes): **4096**

**Cache Performance**

Memory Access Count: **406203** Cache Block Table (block 0 at top)

Cache Hit Count: **276247** ☐ = empty

Cache Miss Count: **129956** ☒ = hit

Cache Hit Rate: **68%** ☐ = miss

**Runtime Log**

☐ Enabled

**Tool Control**

**Disconnect from MIPS** **Reset** **Close**

Figura 6-4: Test de caché: Selección ID = 04

**Data Cache Simulation Tool, Version 1.2**

### Simulate and illustrate data cache performance

#### Cache Organization

Placement Policy: **Fully Associative** Number of blocks: **128**

Block Replacement Policy: **LRU** Cache block size (words): **16**

Set size (blocks): **128** Cache size (bytes): **8192**

#### Cache Performance

Memory Access Count: **406203**

Cache Hit Count: **283361**

Cache Miss Count: **122842**

Cache Hit Rate: **70%**

Cache Block Table (block 0 at top)

☐ = empty  
☒ = hit  
☐ = miss

#### Runtime Log

☐ Enabled

#### Tool Control

**Disconnect from MIPS** **Reset** **Close**

Figura 6-5: Test de caché: Selección ID = 05

**Data Cache Simulation Tool, Version 1.2**


---

**Simulate and illustrate data cache performance**

**Cache Organization**

Placement Policy	N-way Set Associative ▼	Number of blocks	128 ▼
Block Replacement Policy	LRU ▼	Cache block size (words)	64 ▼
Set size (blocks)	16 ▼	Cache size (bytes)	32768

**Cache Performance**

Memory Access Count	406203	<b>Cache Block Table</b> (block 0 at top)  <input type="checkbox"/> = empty <input type="checkbox"/> = hit <input type="checkbox"/> = miss	
Cache Hit Count	302335		
Cache Miss Count	103868		
Cache Hit Rate	74%		

**Runtime Log**

☐ Enabled

**Tool Control**

Disconnect from MIPSResetClose

Figura 6-6: Test de caché: Selección ID = 06



**Data Cache Simulation Tool, Version 1.2**

### Simulate and illustrate data cache performance

**Cache Organization**

Placement Policy	Direct Mapping	Number of blocks	128
Block Replacement Policy	LRU	Cache block size (words)	2048
Set size (blocks)	1	Cache size (bytes)	1048576

**Cache Performance**

Memory Access Count	406203	Cache Block Table (block 0 at top)  <input type="checkbox"/> = empty <input type="checkbox"/> = hit <input type="checkbox"/> = miss
Cache Hit Count	406167	
Cache Miss Count	36	
Cache Hit Rate	100%	

**Runtime Log**

☐ Enabled

**Tool Control**

Disconnect from MIPS      Reset      Close

*Figura 6-7: Test de caché: Selección ID = 07*

**Data Cache Simulation Tool, Version 1.2**

---

**Simulate and illustrate data cache performance**

**Cache Organization**

Placement Policy	Fully Associative	Number of blocks	128
Block Replacement Policy	LRU	Cache block size (words)	2048
Set size (blocks)	128	Cache size (bytes)	1048576

**Cache Performance**

Memory Access Count	406203	<b>Cache Block Table</b> (block 0 at top)  <input type="checkbox"/> = empty <input style="color: green; font-weight: bold; font-size: 1.2em; vertical-align: middle;"/> = hit <input style="color: red; font-weight: bold; font-size: 1.2em; vertical-align: middle;"/> = miss
Cache Hit Count	406169	
Cache Miss Count	34	
Cache Hit Rate	100%	

**Runtime Log**

☐ Enabled

**Tool Control**

Disconnect from MIPSResetClose

Figura 6-8: Test de caché: Selección ID = 08

**Data Cache Simulation Tool, Version 1.2**

### Simulate and illustrate data cache performance

#### Cache Organization

Placement Policy: **N-way Set Associative** Number of blocks: **128**

Block Replacement Policy: **LRU** Cache block size (words): **2048**

Set size (blocks): **16** Cache size (bytes): **1048576**

#### Cache Performance

Memory Access Count: **406203** Cache Block Table (block 0 at top)

Cache Hit Count: **406168** ☐ = empty

Cache Miss Count: **35** ☒ = hit

Cache Hit Rate: **100%** ☐ = miss

#### Runtime Log

☐ Enabled

#### Tool Control

**Disconnect from MIPS** **Reset** **Close**

*Figura 6-9: Test de caché: Selección ID = 09*

**Data Cache Simulation Tool, Version 1.2**

### Simulate and illustrate data cache performance

#### Cache Organization

Placement Policy: **Direct Mapping** Number of blocks: **8**

Block Replacement Policy: **LRU** Cache block size (words): **4**

Set size (blocks): **1** Cache size (bytes): **128**

#### Cache Performance

Memory Access Count: **3310682**

Cache Hit Count: **92766**

Cache Miss Count: **3217916**

Cache Hit Rate: **3%**

Cache Block Table (block 0 at top)

- ☐ = empty
- ☒ = hit
- ☒ = miss

Runtime Log

☐ Enabled

Tool Control

**Disconnect from MIPS** **Reset** **Close**

*Figura 6-10: Test de caché: Mezcla ID = 01*

**Data Cache Simulation Tool, Version 1.2**

---

**Simulate and illustrate data cache performance**

**Cache Organization**

Placement Policy	Fully Associative	Number of blocks	8
Block Replacement Policy	LRU	Cache block size (words)	4
Set size (blocks)	8	Cache size (bytes)	128

**Cache Performance**

Memory Access Count	3310682	<b>Cache Block Table</b> (block 0 at top) <input type="checkbox"/> = empty <input checked="" type="checkbox"/> = hit <input checked="" type="checkbox"/> = miss
Cache Hit Count	81717	
Cache Miss Count	3228965	
Cache Hit Rate	2%	

**Runtime Log**

☐ Enabled

**Tool Control**

Disconnect from MIPS      Reset      Close

Figura 6-11: Test de caché: Mezcla ID = 02

**Data Cache Simulation Tool, Version 1.2**

### Simulate and illustrate data cache performance

#### Cache Organization

Placement Policy: **N-way Set Associative** Number of blocks: **64**

Block Replacement Policy: **LRU** Cache block size (words): **4**

Set size (blocks): **8** Cache size (bytes): **1024**

#### Cache Performance

Memory Access Count: **3310682**

Cache Hit Count: **170005**

Cache Miss Count: **3140677**

Cache Hit Rate: **5%**

Cache Block Table (block 0 at top)

☐ = empty  
☐ = hit  
☐ = miss

#### Runtime Log

☐ Enabled

#### Tool Control

**Disconnect from MIPS** **Reset** **Close**

*Figura 6-12: Test de caché: Mezcla ID = 03*

**Data Cache Simulation Tool, Version 1.2**

### Simulate and illustrate data cache performance

#### Cache Organization

Placement Policy: **Direct Mapping** Number of blocks: **64**

Block Replacement Policy: **LRU** Cache block size (words): **16**

Set size (blocks): **1** Cache size (bytes): **4096**

#### Cache Performance

Memory Access Count: **3310682**

Cache Hit Count: **266219**

Cache Miss Count: **3044463**

Cache Hit Rate: **8%**

Cache Block Table (block 0 at top)

☐ = empty  
☒ = hit  
☐ = miss

#### Runtime Log

☐ Enabled

#### Tool Control

**Disconnect from MIPS** **Reset** **Close**

Figura 6-13: Test de caché: Mezcla ID = 04

**Data Cache Simulation Tool, Version 1.2**

### Simulate and illustrate data cache performance

#### Cache Organization

Placement Policy: **Fully Associative** Number of blocks: **128**

Block Replacement Policy: **LRU** Cache block size (words): **16**

Set size (blocks): **128** Cache size (bytes): **8192**

#### Cache Performance

Memory Access Count: **3310682**

Cache Hit Count: **365832**

Cache Miss Count: **2944850**

Cache Hit Rate: **11%**

Cache Block Table (block 0 at top)

☐ = empty  
☐ = hit  
☐ = miss

#### Runtime Log

☐ Enabled

#### Tool Control

**Disconnect from MIPS** **Reset** **Close**

*Figura 6-14: Test de caché: Mezcla ID = 05*



**Data Cache Simulation Tool, Version 1.2**

### Simulate and illustrate data cache performance

**Cache Organization**

Placement Policy: **N-way Set Associative** Number of blocks: **128**

Block Replacement Policy: **LRU** Cache block size (words): **64**

Set size (blocks): **16** Cache size (bytes): **32768**

**Cache Performance**

Memory Access Count: **3310682** Cache Block Table (block 0 at top)

Cache Hit Count: **713550** ☐ = empty

Cache Miss Count: **2597132** ☒ = hit

Cache Hit Rate: **22%** ☐ = miss

**Runtime Log**

☐ Enabled

**Tool Control**

**Disconnect from MIPS** **Reset** **Close**

Figura 6-15: Test de caché: Mezcla ID = 06

**Data Cache Simulation Tool, Version 1.2**

### Simulate and illustrate data cache performance

#### Cache Organization

Placement Policy: **Direct Mapping** Number of blocks: **128**

Block Replacement Policy: **LRU** Cache block size (words): **2048**

Set size (blocks): **1** Cache size (bytes): **1048576**

#### Cache Performance

Memory Access Count: **3310682**

Cache Hit Count: **3310644**

Cache Miss Count: **38**

Cache Hit Rate: **100%**

Cache Block Table (block 0 at top)

☐ = empty  
☒ = hit  
☐ = miss

#### Runtime Log

☐ Enabled

#### Tool Control

**Disconnect from MIPS** **Reset** **Close**

*Figura 6-16: Test de caché: Mezcla ID = 07*

**Data Cache Simulation Tool, Version 1.2**

### Simulate and illustrate data cache performance

#### Cache Organization

Placement Policy: **Fully Associative** Number of blocks: **128**

Block Replacement Policy: **LRU** Cache block size (words): **2048**

Set size (blocks): **128** Cache size (bytes): **1048576**

#### Cache Performance

Memory Access Count: **3310682**

Cache Hit Count: **3310646**

Cache Miss Count: **36**

Cache Hit Rate: **100%**

Cache Block Table (block 0 at top)

☐ = empty  
☒ = hit  
☐ = miss

#### Runtime Log

☐ Enabled

#### Tool Control

**Disconnect from MIPS** **Reset** **Close**

*Figura 6-17: Test de caché: Mezcla ID = 08*

**Data Cache Simulation Tool, Version 1.2**

### Simulate and illustrate data cache performance

#### Cache Organization

Placement Policy: **N-way Set Associative** Number of blocks: **128**

Block Replacement Policy: **LRU** Cache block size (words): **2048**

Set size (blocks): **16** Cache size (bytes): **1048576**

#### Cache Performance

Memory Access Count: **3310682**

Cache Hit Count: **3310647**

Cache Miss Count: **35**

Cache Hit Rate: **100%**

Cache Block Table (block 0 at top)

☐ = empty  
☐ = hit  
☐ = miss

#### Runtime Log

☐ Enabled

#### Tool Control

**Disconnect from MIPS** **Reset** **Close**

*Figura 6-18: Test de caché: Mezcla ID = 09*

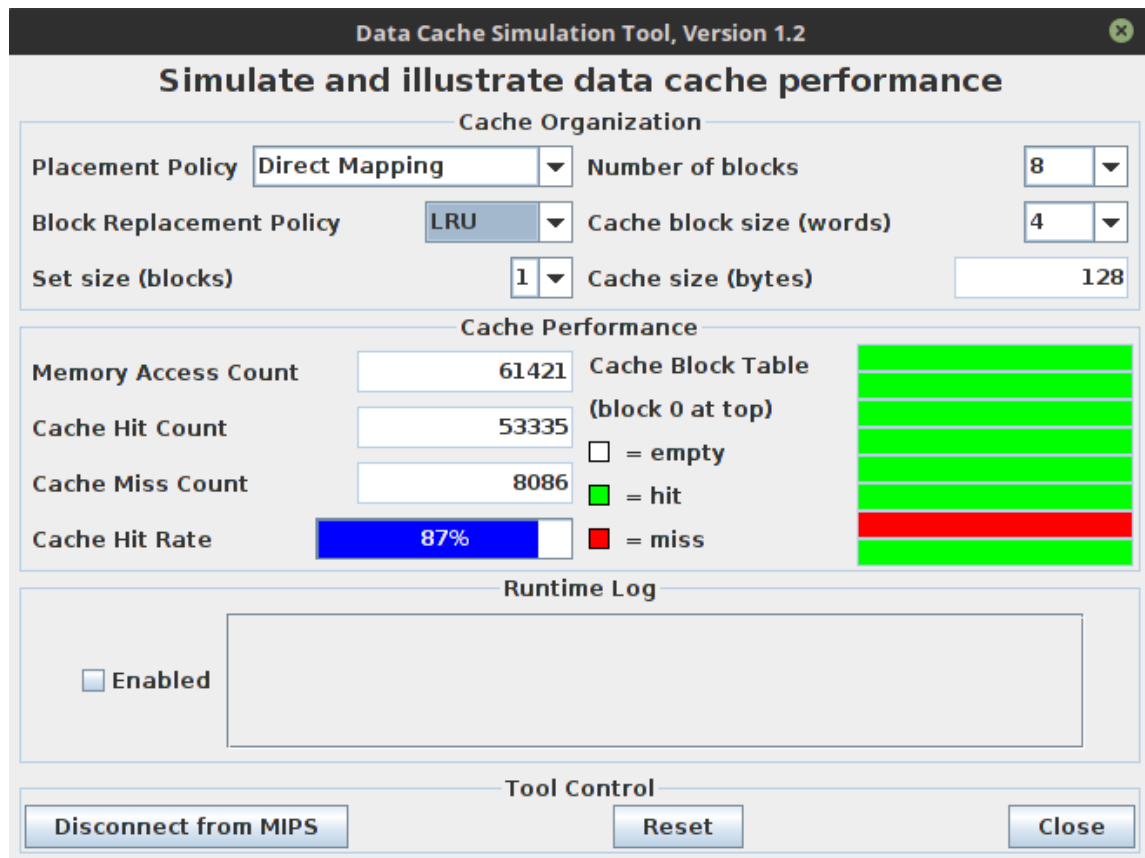


Figura 6-19: Test de caché: Programa con arreglos ID = 01

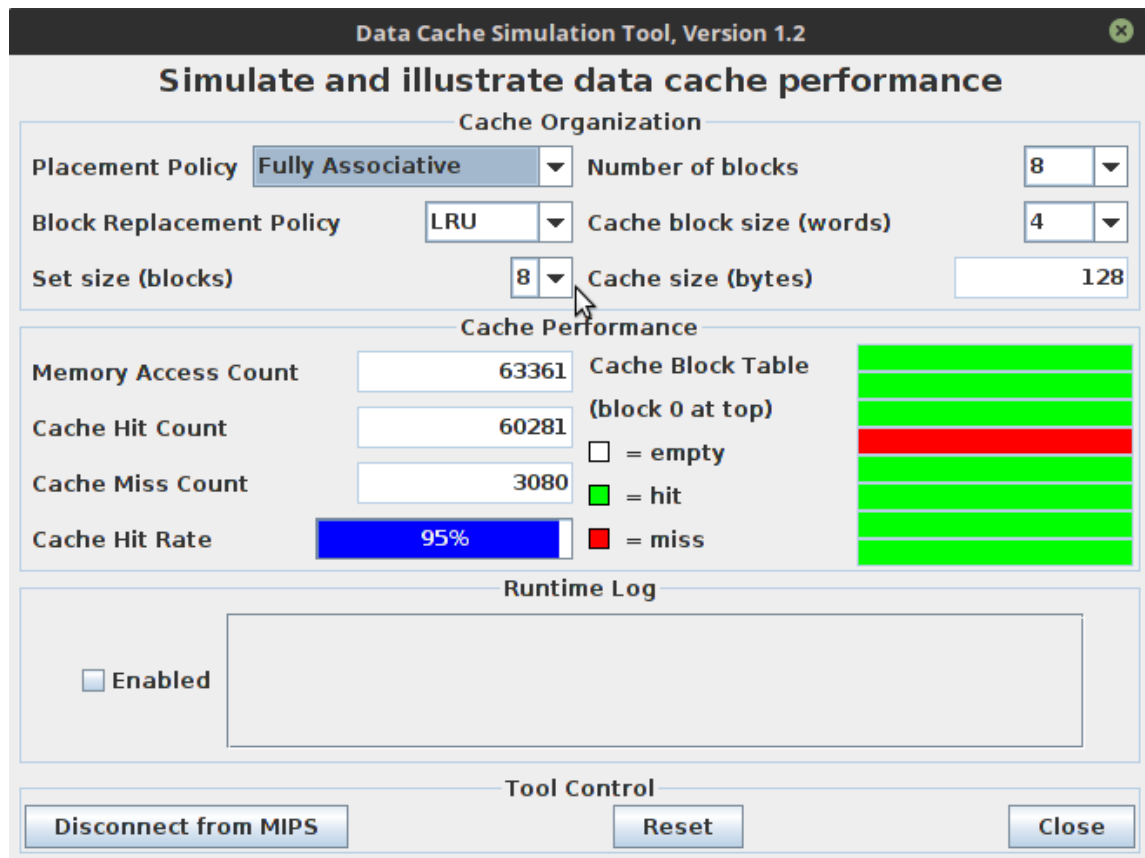


Figura 6-20: Test de caché: Programa con arreglos ID = 02

**Data Cache Simulation Tool, Version 1.2**

### Simulate and illustrate data cache performance

**Cache Organization**

Placement Policy: **N-way Set Associative** | Number of blocks: **64**

Block Replacement Policy: **LRU** | Cache block size (words): **4**

Set size (blocks): **8** | Cache size (bytes): **1024**

**Cache Performance**

Memory Access Count: **61677** | Cache Block Table (block 0 at top)

Cache Hit Count: **60599** | ☐ = empty

Cache Miss Count: **1078** | ☒ = hit

Cache Hit Rate: **98%** | ☐ = miss

**Runtime Log**

☐ Enabled

**Tool Control**

**Disconnect from MIPS** | **Reset** | **Close**

Figura 6-21: Test de caché: Programa con arreglos ID = 03

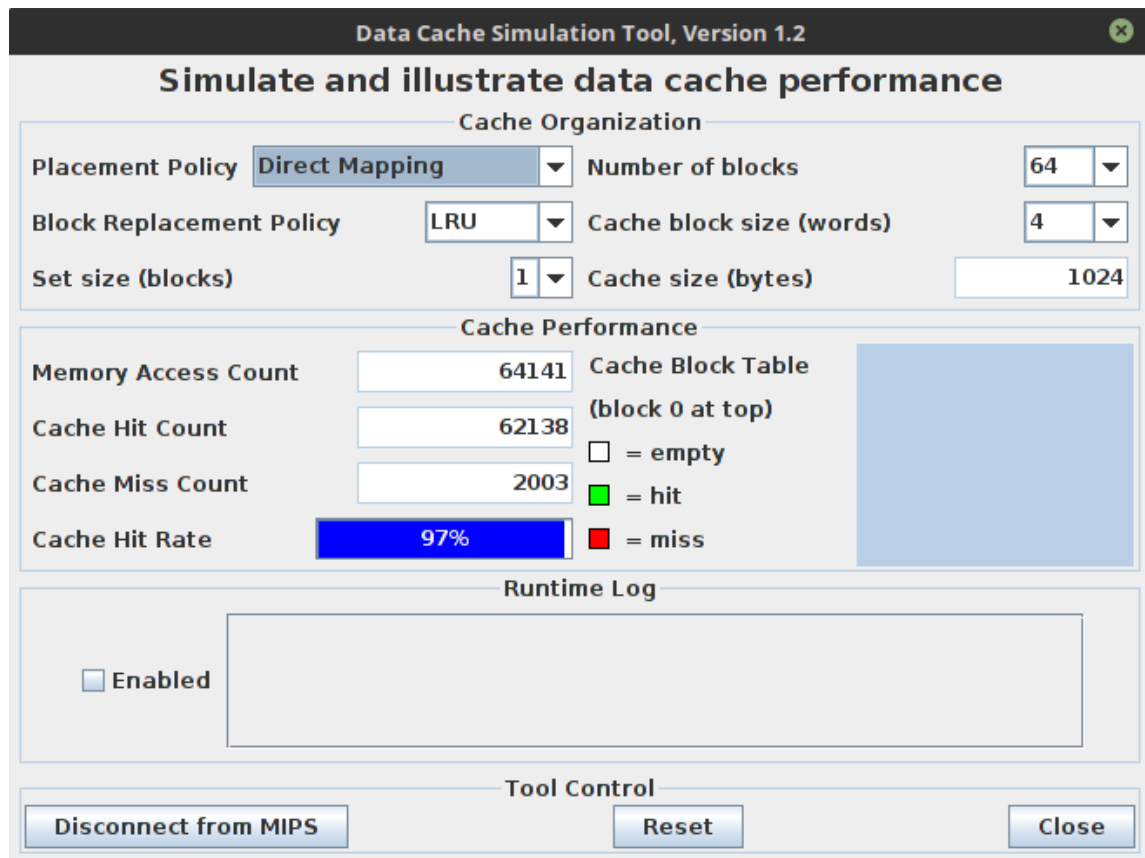


Figura 6-22: Test de caché: Programa con arreglos ID = 04



**Data Cache Simulation Tool, Version 1.2**

### Simulate and illustrate data cache performance

**Cache Organization**

Placement Policy: **Fully Associative** Number of blocks: **128**

Block Replacement Policy: **LRU** Cache block size (words): **16**

Set size (blocks): **128** Cache size (bytes): **8192**

**Cache Performance**

Memory Access Count: **60673** Cache Block Table (block 0 at top)

Cache Hit Count: **60598** ☐ = empty

Cache Miss Count: **75** ☒ = hit

Cache Hit Rate: **100%** ☐ = miss

**Runtime Log**

☐ Enabled

**Tool Control**

**Disconnect from MIPS** **Reset** **Close**

Figura 6-23: Test de caché: Programa con arreglos ID = 05

**Data Cache Simulation Tool, Version 1.2**

### Simulate and illustrate data cache performance

**Cache Organization**

Placement Policy N-way Set Associative Number of blocks 128

Block Replacement Policy LRU Cache block size (words) 64

Set size (blocks) 16 Cache size (bytes) 32768

**Cache Performance**

Memory Access Count 64141 Cache Block Table (block 0 at top)

Cache Hit Count 64117 ☐ = empty

Cache Miss Count 24 ☒ = hit

Cache Hit Rate 100% ☐ = miss

**Runtime Log**

☐ Enabled

**Tool Control**

Disconnect from MIPS Reset Close

Figura 6-24: Test de caché: Programa con arreglos ID = 06

**Data Cache Simulation Tool, Version 1.2**

---

**Simulate and illustrate data cache performance**

**Cache Organization**

Placement Policy	Direct Mapping	Number of blocks	8
Block Replacement Policy	LRU	Cache block size (words)	4
Set size (blocks)	1	Cache size (bytes)	128

**Cache Performance**

Memory Access Count	60257	Cache Block Table (block 0 at top) <input type="checkbox"/> = empty <input checked="" type="checkbox"/> = hit <input type="checkbox"/> = miss	<div></div>
Cache Hit Count	51818		<div></div>
Cache Miss Count	8439		<div></div>
Cache Hit Rate	86%		<div></div>

**Runtime Log**

☐ Enabled

**Tool Control**

Disconnect from MIPS      Reset      Close

Figura 6-25: Test de caché: Programa con arreglos con mejora ID = 01

**Data Cache Simulation Tool, Version 1.2**

### Simulate and illustrate data cache performance

**Cache Organization**

Placement Policy Fully Associative Number of blocks 8

Block Replacement Policy LRU Cache block size (words) 4

Set size (blocks) 8 Cache size (bytes) 128

**Cache Performance**

Memory Access Count 60183 Cache Block Table (block 0 at top)

Cache Hit Count 56987 ☐ = empty

Cache Miss Count 3196 ☒ = hit

Cache Hit Rate 95% ☒ = miss

**Runtime Log**

☐ Enabled

**Tool Control**

Disconnect from MIPS Reset Close

Figura 6-26: Test de caché: Programa con arreglos con mejora ID = 02

**Data Cache Simulation Tool, Version 1.2**

---

**Simulate and illustrate data cache performance**

**Cache Organization**

Placement Policy	N-way Set Associative	Number of blocks	64
Block Replacement Policy	LRU	Cache block size (words)	4
Set size (blocks)	8	Cache size (bytes)	1024

**Cache Performance**

Memory Access Count	60015	<b>Cache Block Table</b> (block 0 at top)  <input type="checkbox"/> = empty <input style="color: green;"/> = hit <input style="color: red;"/> = miss
Cache Hit Count	58350	
Cache Miss Count	1665	
Cache Hit Rate	97%	

**Runtime Log**

☐ Enabled

**Tool Control**

Disconnect from MIPSResetClose

Figura 6-27: Test de caché: Programa con arreglos con mejora ID = 03

**Data Cache Simulation Tool, Version 1.2**

### Simulate and illustrate data cache performance

**Cache Organization**

Placement Policy: **Direct Mapping** Number of blocks: **64**

Block Replacement Policy: **LRU** Cache block size (words): **16**

Set size (blocks): **1** Cache size (bytes): **4096**

**Cache Performance**

Memory Access Count: **60181** Cache Block Table (block 0 at top)

Cache Hit Count: **59339** ☐ = empty

Cache Miss Count: **842** ☒ = hit

Cache Hit Rate: **99%** ☐ = miss

**Runtime Log**

☐ Enabled

**Tool Control**

**Disconnect from MIPS** **Reset** **Close**

Figura 6-28: Test de caché: Programa con arreglos con mejora ID = 04

**Data Cache Simulation Tool, Version 1.2**

### Simulate and illustrate data cache performance

**Cache Organization**

Placement Policy: **Fully Associative** Number of blocks: **128**

Block Replacement Policy: **LRU** Cache block size (words): **16**

Set size (blocks): **128** Cache size (bytes): **8192**

**Cache Performance**

Memory Access Count: **60183** Cache Block Table (block 0 at top)

Cache Hit Count: **59734** ☐ = empty

Cache Miss Count: **449** ☒ = hit

Cache Hit Rate: **99%** ☐ = miss

**Runtime Log**

☐ Enabled

**Tool Control**

**Disconnect from MIPS** **Reset** **Close**

Figura 6-29: Test de caché: Programa con arreglos con mejora ID = 05

**Data Cache Simulation Tool, Version 1.2**

### Simulate and illustrate data cache performance

**Cache Organization**

Placement Policy: **N-way Set Associative** | Number of blocks: **128**

Block Replacement Policy: **LRU** | Cache block size (words): **64**

Set size (blocks): **16** | Cache size (bytes): **32768**

**Cache Performance**

Memory Access Count: **60183** | Cache Block Table (block 0 at top)

Cache Hit Count: **59886** | ☐ = empty

Cache Miss Count: **297** | ☒ = hit

Cache Hit Rate: **100%** | ☐ = miss

**Runtime Log**

☐ Enabled

**Tool Control**

**Disconnect from MIPS** | **Reset** | **Close**

Figura 6-30: Test de caché: Programa con arreglos con mejora ID = 06



## CAPÍTULO 7. BIBLIOGRAFÍA

- [1] F. Garay, *Enunciado laboratorio 3*, 2016. [Online]. Available: <http://www.udesantiagovirtual.cl/moodle2/mod/resource/view.php?id=115596>.
- [2] D. K. Vollmar, *Mars - mips assembly and runtime simulator*, 2014. [Online]. Available: <http://courses.missouristate.edu/KenVollmar/mars/>.
- [3] N. O. González, *Repositorio de trabajo - lab3\_orga*, 2016. [Online]. Available: [https://github.com/pastordrogo/lab3\\_orga](https://github.com/pastordrogo/lab3_orga).
- [4] F. I. Universidad Complutense de Madrid, *Estructura de computadores*, 2015. [Online]. Available: <http://www.fdi.ucm.es/profesor/jjruiz/web2/temas/ec6.pdf>.
- [5] Wikipedia, *Eficiencia algorítmica - wikipedia*, 2016. [Online]. Available: [https://es.wikipedia.org/wiki/Eficiencia\\_Algor%C3%ADmica](https://es.wikipedia.org/wiki/Eficiencia_Algor%C3%ADmica).
- [6] Wikipedia, *Ordenamiento por selección - wikipedia*, 2015. [Online]. Available: [https://es.wikipedia.org/wiki/Ordenamiento\\_por\\_selecci%C3%B3n](https://es.wikipedia.org/wiki/Ordenamiento_por_selecci%C3%B3n).
- [7] Wikipedia, *Ordenamiento por mezcla - wikipedia*, 2015. [Online]. Available: [https://es.wikipedia.org/wiki/Ordenamiento\\_por\\_mezcla#Optimizando\\_merge\\_sort](https://es.wikipedia.org/wiki/Ordenamiento_por_mezcla#Optimizando_merge_sort).
- [8] Desconocido, *Algoritmos recursivos de ordenamiento*. [Online]. Available: <https://docs.google.com/document/d/1S1neE40ehnlEHOFRHW2XSsc0ddmvEyP3qWn-4EnYqP4/edit>.
- [9] T. G. Foundation, *Estructuras de datos: Listas enlazadas, pilas y colas*, 2002. [Online]. Available: <http://www.calcifer.org/documentos/librognome/glib-lists-queues.html>.
- [10] S. P. .-. C. C. clase, *Estructuras de datos: Capítulo 5 listas doblemente enlazadas*, 2001. [Online]. Available: <http://c.conclase.net/edd/?cap=005>.
- [11] S. P. .-. C. C. clase, *Estructuras de datos: Capítulo 4 listas circulares*, 2001. [Online]. Available: <http://c.conclase.net/edd/?cap=004>.
- [12] J. César, *La clase vector y sus métodos más importantes en c++*, 2014. [Online]. Available: <https://blogdelingeniero1.wordpress.com/2014/07/22/la-clase-vector-y-sus-metodos-mas-importantes-en-cpp-c/>.
- [13] cplusplus, *Vector*, 2016. [Online]. Available: <http://www.cplusplus.com/reference/vector/vector/>.
- [14] Wikipedia, *Fuzz testing - wikipedia*, 2016. [Online]. Available: [https://en.wikipedia.org/wiki/Fuzz\\_testing](https://en.wikipedia.org/wiki/Fuzz_testing).
- [15] Wikipedia, *Orden total - wikipedia*, 2016. [Online]. Available: [https://es.wikipedia.org/wiki/Orden\\_total](https://es.wikipedia.org/wiki/Orden_total).

- [16] E. R. Olivo, *Organizacion de computadores captulo iii: Rendimiento de caché*, 2015. [Online]. Available: [http://www.udesantiagovirtual.cl/moodle2/pluginfile.php?file=%2F111759%2Fmod\\_resource%2Fcontent%2F1%2Fmem\\_2.pdf](http://www.udesantiagovirtual.cl/moodle2/pluginfile.php?file=%2F111759%2Fmod_resource%2Fcontent%2F1%2Fmem_2.pdf).
- [17] E. R. Olivo, *Organizacion de computadores captulo iii: Jerarquia de memoria y caché*, 2015. [Online]. Available: [http://www.udesantiagovirtual.cl/moodle2/pluginfile.php?file=%2F111758%2Fmod\\_resource%2Fcontent%2F1%2Fmem\\_1.pdf](http://www.udesantiagovirtual.cl/moodle2/pluginfile.php?file=%2F111758%2Fmod_resource%2Fcontent%2F1%2Fmem_1.pdf).
- [18] E. Bahit, *Capítulo 10. un paseo por los módulos de la librería estándar - 10.1 módulos de sistema*, 2013. [Online]. Available: [http://librosweb.es/libro/python/capitulo\\_10/modulos\\_de\\_sistema.html](http://librosweb.es/libro/python/capitulo_10/modulos_de_sistema.html).
- [19] Wikipedia, *Prueba de los rangos con signo de wilcoxon - wikipedia*, 2016. [Online]. Available: [https://es.wikipedia.org/wiki/Prueba\\_de\\_los\\_rangos\\_con\\_signo\\_de\\_Wilcoxon](https://es.wikipedia.org/wiki/Prueba_de_los_rangos_con_signo_de_Wilcoxon).