Examen de Arquitectura de Computadores 1
Josafat Vargas - 2013030892
Victor Montero - 2013014152

Parte I

Punto 1

- El **call stack** es un mecanismo que utiliza un intérprete para saber dónde se encuentra en un código que llama a varias funciones, qué función se está ejecutando y que otras funciones fueran llamadas dentro de esta. [1.1.1]
- Es la lista de nombres de métodos, llamados durante la ejecución hasta el que se está ejecutando actualmente, actúa como una forma de debugueo para una aplicación que puede ser llamada desde diferentes contextos. [1.1.2]
- Es una estructura de datos que utiliza el principio de LIFO para manejar de forma temporal la llamada de funciones. [1.1.3]

[1.1.1] https://developer.mozilla.org/en-US/docs/Glossary/Call_stack
[1.1.2] https://www.techopedia.com/definition/25586/call-stack-c
[1.1.3] https://www.freecodecamp.org/news/understanding-the-javascript-call-stack-861e41ae61d4/

El procedimiento para el llamado de funciones es el siguiente [1.1.4]:

1. Se debe preservar el registro ebp (frame pointer) entre llamadas. Si la función modifica este registro, entonces su valor debe ser guardado en el stack.
2. El registro rbp apunta al frame de la función.
3. Se hace espacio en el stack para las variables locales, restando posiciones al puntero.
4. Se realiza el almacenamiento de variables con push.
5. Se realizan las operaciones de la función.
6. El registro eax almacena el resultado.
7. Se deja la función, poniendo el valor del registro ebp en esp.
8. Se extrae el valor del registro ebp del stack.
9. Se limpia el stack de variables locales
10. Se retorna a la función previa.

[1.1.4]https://eli.thegreenplace.net/2011/02/04/where-the-top-of-the-stack-is-on-x86/#id6

Punto 2

Las variables estáticas tienen la propiedad de preservar su valor incluso cuando están fuera de su alcance, de esta forma, las variables solo requieren inicializarse una vez en el programa y se mantienen en memoria mientras el programa está corriendo. Una variable normal o automática es destruida cuando la función en la que fue llamada fue inicializada finaliza, pero en un ambiente estático, si la variable se declara, mantiene el valor que tenía cuando fue inicializada originalmente (o el valor que tomó durante la ejecución). [1.2.1]

Las variables estáticas se almacenan en el segmento de datos, no en en el segmento del stack y deben ser inicializadas usando constantes literales, si el programador no la inicializa, toman automáticamente el valor de cero, en vez de un valor basura como ocurre en las variables no estáticas. [1.2.1]

[1.2.1] https://www.geeksforgeeks.org/static-variables-in-c/

Punto 3

ELF es la abreviación de *Executable and Linkable Format* y define la estructura de binarios, bibliotecas y archivos nucleares[1.3.1]. La especificación formal le permite al sistema operativo interpretar su código de máquina fundamental, de forma correcta. Los archivos ELF son, típicamente, la salida de un compilador o linker y tienen un formato binario. El archivo consiste de un header y datos de archivo[1.3.2]:

Header

El header comienza con ***magia*** que provee información acerca del archivo.

1. **Definición del campo de clase**. Este valor determina para qué tipo de arquitectura es el archivo. Puede ser 01 para 32-bits or 02 para 64-bits.
2. La siguiente parte es el **campo de data** y puede tener dos opciones: 01 para Little-endian o 02 para big-endian.
3. Siguiente en la magia, es un 01 que es significa **versión** 1, la única disponible en este momento.
4. **Interfaz de Aplicación Binaria** (ABI): De esta forma tanto el sistema operativo como las aplicaciones saben qué esperar y las funciones son correctamente reenviadas.
5. Cuando es necesario, se puede especificar una **versión del ABI.**
6. **Tipo de máquina** esperada
7. **Campo de tipo** dice para que fue creado el archivo. Hay algunos tipos comunes:
    a. CORE (valor 4)
    b. DYN (Archivo de objetos compartidos), para bibliotecas (valor 3)
    c. EXEC (Archivo ejecutable), para binarios (valor 2)
    d. REL (Archivo relocable), antes de vincular a un archivo ejecutable (valor 1)

Datos del archivo

1. **Headers del programa.** Un archivo ELF consiste de cero o más segmentos y describe cómo crear un proceso o imágen de memoria para tiempo de ejecución. Cuando el kernel ve estos segmentos los usa para mapearlos a un espacio de direcciones de memoria virtual.
2. **Headers de sección.** Esta sección define las otras secciones del archivo, esta se utiliza para vinculaciones y recolocaciones.
3. **Data.** Los datos del archivo, contiene secciones como:
    a. .text: código ejecutable.
    b. .data: data inicializada con permisos tanto de escritura como de lectura.
    c. .rodata: data inicializada con solo permisos de lectura.
    d. .bss: datos sin inicializar, con permisos de escritura y lectura.

[1.3.1] https://linux-audit.com/elf-binaries-on-linux-understanding-and-analysis/
[1.3.2] https://ocw.cs.pub.ro/courses/cns/labs/lab-03

Punto 4

En la figura 1 se muestra el código ASM resultado de la compilación de la función sin variables *static* usando gcc.

Parte I ▸ ASM notStatic.s

```
1     .file   "notStatic.c"
2     .text
3     .globl  method_without_static_variables
4     .type   method_without_static_variables, @function
5   method_without_static_variables:
6       pushq   %rbp
7       movq    %rsp, %rbp
8       movl    %edi, -20(%rbp)
9       movl    %esi, -24(%rbp)
10      movl    %edx, -28(%rbp)
11      addl    $1, -4(%rbp)
12      movl    -20(%rbp), %edx
13      movl    -24(%rbp), %eax
14      addl    %eax, %edx
15      movl    -28(%rbp), %eax
16      addl    %eax, %edx
17      movl    -4(%rbp), %eax
18      addl    %edx, %eax
19      popq    %rbp
20      ret
21      .size   method_without_static_variables, .-method_without_static_variables
22      .ident  "GCC: (Ubuntu 8.3.0-6ubuntu1~18.10) 8.3.0"
23      .section    .note.GNU-stack,"",@progbits
24
```

Figura 1.4.1. ASM generado para “method_without_static_variables”

Las líneas 1 a 4 son directivas del procesador para establecer aspectos como los espacios en memoria.
Las líneas 6 y 7 son el punto de entrada de la función. La línea 6 almacena el valor del stack frame. En la línea 7 se indica que el stack pointer debe apuntar al top del stack.
A partir de la línea 8 hasta la 18 se encuentra la traducción de las instrucciones del método. En estas se puede ver cómo se utilizan posiciones relativas al registro rbp, es decir se utilizan las variables almacenadas en el stack para llenar los registros que son operados. Por ejemplo la posición rbp - 4 corresponde a la variable “counter” del codigo C.
En las líneas 19 y 20 se puede observar la secuencia de salida . En la línea 19 se restaura el valor del frame pointer que se encuentra en el top del stack. En la línea 20 se utiliza el comando ret para regresar a la función que llama.
Las líneas 21 a 23 son más directivas para indicar tamaños de datos, comentarios y la sección utilizada.

En la figura 2 se muestra el código ASM resultado de la compilación de la función con variables *static* usando gcc.

Parte I ▸ static.s

```
1      .file   "static.c"
2      .text
3      .globl  method_with_static_variables
4      .type   method_with_static_variables, @function
5  method_with_static_variables:
6      pushq   %rbp
7      movq    %rsp, %rbp
8      movl    %edi, -4(%rbp)
9      movl    %esi, -8(%rbp)
10     movl    %edx, -12(%rbp)
11     movl    counter.1960(%rip), %eax
12     addl    $1, %eax
13     movl    %eax, counter.1960(%rip)
14     movl    -4(%rbp), %edx
15     movl    -8(%rbp), %eax
16     addl    %eax, %edx
17     movl    -12(%rbp), %eax
18     addl    %eax, %edx
19     movl    counter.1960(%rip), %eax
20     addl    %edx, %eax
21     popq    %rbp
22     ret
23     .size   method_with_static_variables, .-method_with_static_variables
24     .local  counter.1960
25     .comm   counter.1960,4,4
26     .ident  "GCC: (Ubuntu 8.3.0-6ubuntu1~18.10) 8.3.0"
27     .section    .note.GNU-stack,"",@progbits
28
```

Figura 1.4.2. ASM generado para “method_with_static_variables”

La estructura del código es muy similar al caso anterior. Las líneas iniciales 1 a 4 son idénticas.
Una vez más en las líneas 6 y 7 se establece el punto de entrada a la función re ubicando el stack frame.
En este caso a partir de la línea 8 que inicia la ejecución de la función se empieza a notar una diferencia respecto al código anterior. La línea 11 presenta la primera diferencia, en esta se ubica el valor del registro eax en un espacio de memoria cuya ubicación se obtiene de forma relativa usando el registro rip y una etiqueta definida en la sección de directivas al final del código. Esto lleva a la siguiente diferencia importante entre ambos códigos y es el uso de las directivas .local y .comm [1.4.1] con las cuales respectivamente se declara un símbolo para la variable declarada como estática en el código C (counter) y se establece el tamaño y alineamiento de dicho símbolo. En resumen, con estas directivas se está realizando la reserva de espacio en el segmento de datos del código para la variable estática, tal y como se mencionó en el punto 2.

Las demás diferencias en la sección de ejecución se presentan en las líneas 13 y 19 en donde a diferencia del primer código se utilizaba como contador un espacio del stack, ahora se utiliza el símbolo de la variable almacenada en la sección de datos del segundo código.

[1.4.1] https://docs.oracle.com/cd/E19253-01/817-5477/eoiyg/index.html

Punto 5

En los archivos ubicados en "ExamenArqui1/Parte1" con nombres "ELFnotStatic.txt" y "ELFStatic.txt" se pueden observar los comandos utilizados para obtener los datos de los archivos ELF, así como los resultados obtenidos. A continuación se presenta un reporte resumiendo los datos más relevantes. En un recuadro se muestra una línea específica del archivo ELF y debajo se hace un análisis comparativo de lo observado.

Header: (Información obtenida usando readelf -h notStatic.o | readelf -h static.o)

Entry point address: 0x0

-Se observa que el "entry point address" del programa en ambos casos es 0x0, esto indica que no se tiene un entry point, lo cual es esperado ya que se compiló sin un main.[1.5.1]

*Start of section headers: 576 (bytes into file) *notStatic* *Number of section headers: 11* *Section header string table index: 10*

*Start of section headers: 704 (bytes into file) *Static* *Number of section headers: 12* *Section header string table index: 11*

-El programa que no hace uso de *static* presenta un menor tamaño de secciones, como se vió en el análisis del código ensamblador en el punto 4, la definición de variables static requiere del uso de espacio en la sección de datos.

Sections: (Información obtenida usando readelf -S notStatic.o | readelf -S static.o)

-La sección de código (.text) presenta un tamaño de 25 en el caso sin static y 33 con static, esto es correspondiente con el aumento en el tamaño de código observado en el programa ensamblador.
-El programa con static tiene una sección extra, como se mencionó antes. Esta sección resultó ser ".rela.text", la cual es utilizada para relacionar la sección de código (".text") con la sección 9, la cual corresponde a ".symtab", que es donde se almacena el símbolo utilizado en el código ensamblador para referirse a la variable estática.

Segments: (Información obtenida usando readelf -l notStatic.o | readelf -l static.o)

There are no program headers in this file.

Esta respuesta se presentó en ambos casos y es debido a que no se cuenta con un main y por lo tanto no se conoce como se agrupan las secciones para ser ejecutadas.

Symbols: (Información obtenida usando readelf -s notStatic.o | readelf -s static.o)

Num:	*Value*	*Size*	*Type*	*Bind*	*Vis*	*Ndx*	*Name*
8:	*0000000000000000*	*37*	*FUNC*	*GLOBAL*	*DEFAULT*	*1*	*method_without_static_var*

**notStatic*

Num:	*Value*	*Size*	*Type*	*Bind*	*Vis*	*Ndx*	*Name*	**static*
9:	*0000000000000000*	*51*	*FUNC*	*GLOBAL*	*DEFAULT*	*1*	*method_with_static_variab*	

-En ambas ejecuciones se obtiene en la última entrada de la tabla de símbolos el nombre del entry point para cada respectiva función, donde además se puede observar un tamaño consistente con la cantidad de instrucciones que tiene cada instrucción

Num:	*Value*	*Size*	*Type*	*Bind*	*Vis*	*Ndx*	*Name*	**static*
5:	0000000000000000	4	OBJECT	LOCAL	DEFAULT	4	counter.1960	

-En la ejecución para la función con static se puede observar una entrada extra en la tabla de símbolos, la cual corresponde al objeto definido para almacenar la variable estática en el código de ensamblador, se puede constatar el nombre de esta variable, así como su tamaño y alineamiento.

[1.5.1] https://www.uclibc.org/docs/elf-64-gen.pdf
[1.5.2] https://ocw.cs.pub.ro/courses/cns/labs/lab-03

Punto 6

Para resolver lo solicitado en este punto se realizaron 4 archivos adicionales. Los archivos notStaticExeTime.c y staticExeTime.c contienen una prueba para evaluar el tiempo de ejecución de la función sin y con variable estática respectivamente. Los archivos notStaticMemConsum.c y staticMemConsum.c son utilizados para el análisis de consumo de memoria.

La prueba de tiempo consiste en un bucle que ejecuta la función 2^30 veces, este valor fue determinado mediante prueba y error, buscando la cantidad máxima de ejecuciones soportadas antes de que el programa fallara. Antes de iniciar el bucle se obtiene el tiempo con la función clock() brindada en la biblioteca time.h y una vez más al finalizar el for. Para obtener el dato de duración en segundos de este

bucle se resta el tiempo dado al final de inicial y se divide entre los ciclos por segundo. Finalmente para obtener el dato de tiempo de ejecución de la función se divide el resultado anterior entre el número de ejecuciones. Los resultados obtenidos se presentan a continuación

	Función sin variable estática	**Función con variable estática**
Tiempo de ejecución (s)	0.000000002432976	0.000000002388101
Tiempo de ejecución (ns)	2.432976	2.388101

Tabla 1.6.1. Resultados de pruebas de tiempo de ejecución

En esta prueba se evidencia una ligera diferencia entre los tiempos de ejecución de las funciones, sin embargo esta es de tan solo 0.044 ns, lo cual no es determinante para establecer si el uso de una variable estática en una función sencilla genera una mejora en el desempeño.

El programa para analizar el consumo de memoria fue similar conceptualmente al anterior, sin embargo se hizo un archivo separado ya que se deseaba obtener información de la ejecución de la función de forma aislada y debido a las operaciones para obtener el tiempo de ejecución en la prueba anterior esto no sería posible.

Para el análisis de consumo de memoria se utilizó la herramienta Massif, que forma parte del suite Valgrind. Se decidió usar esta herramienta específicamente debido a que permite medir cuánta memoria heap es utilizada en el programa, así como el tamaño del stack.[1.6.1]

Para la ejecución de la herramienta se utilizaron los siguientes comandos:

Comando	**Descripción**
valgrind --tool=massif --stack=yes --time-unit=B ./binFileName	Con la bandera --time-unit se indica que se desea que la unidad utilizada para medir sea el numero de Byte reservados, ya que por defecto es la cantidad de llamadas a funciones, pero al ser un programa tan sencillo este dato no resulta muy útil. Con la bandera --stack se indica que se desea perfilar el stack, ya que por defecto estos datos no se analizan.
valgrind --tool=massif --heap=yes --time-unit=B ./binFileName	Con la bandera --heap se indica que se desea perfilar el uso del heap.
ms_print massif.out.name	Este comando es utilizado para una visualización sencilla del archivo de salida generado por la herramienta Massif.

Tabla 1.6.2. Comandos de ejecución de Massif de Valgrind

A continuación se presentan los resultados obtenidos:

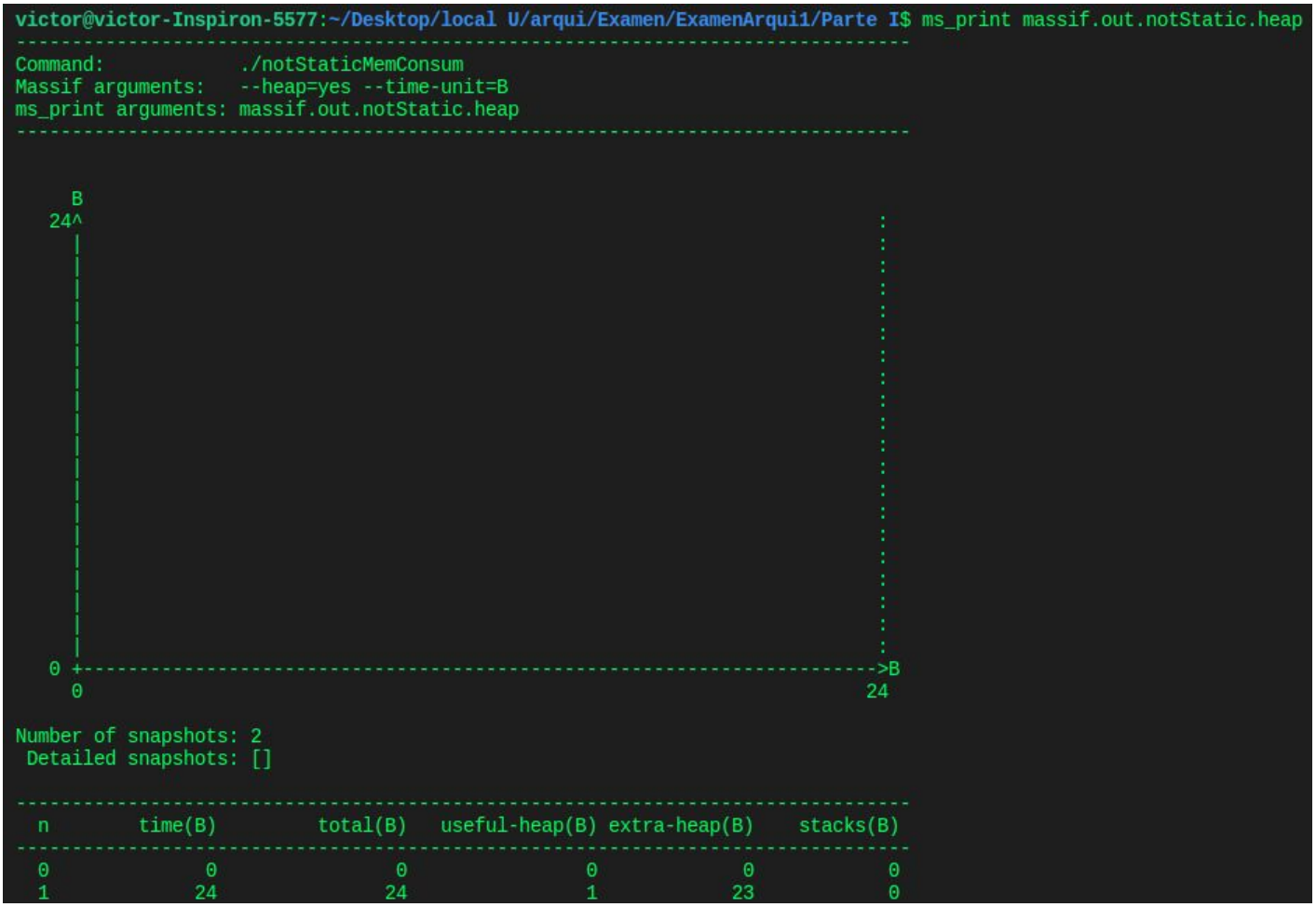

Figura 1.6.1. Salida de análisis del heap con Massif para la función sin variable estática

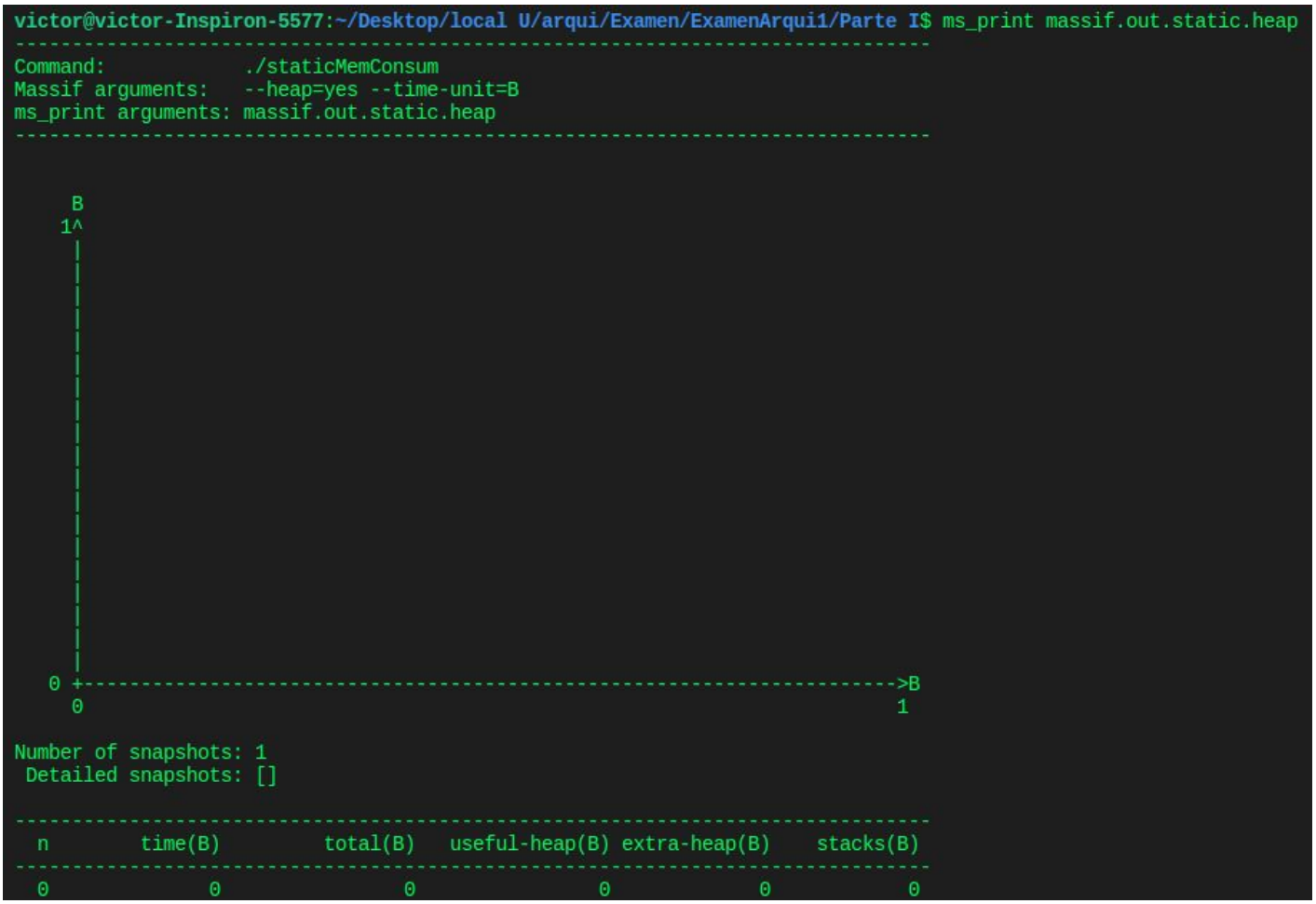

Figura 1.6.2. Salida de análisis del heap con Massif para la función con variable estática

De la figura 1.6.1 se puede notar que al no utilizar una variable estática el programa requiere de un uso de 24B del heap y que cuando se utiliza la variable estática no se hace uso del heap como muestra a figura 1.6.2. Tomando en cuenta que el heap es memoria dinámica que se utiliza en la ejecución del programa, la versión del programa con la variable estática no requiere usar el heap ya que tiene su variable en una sección asignada al iniciar el programa.

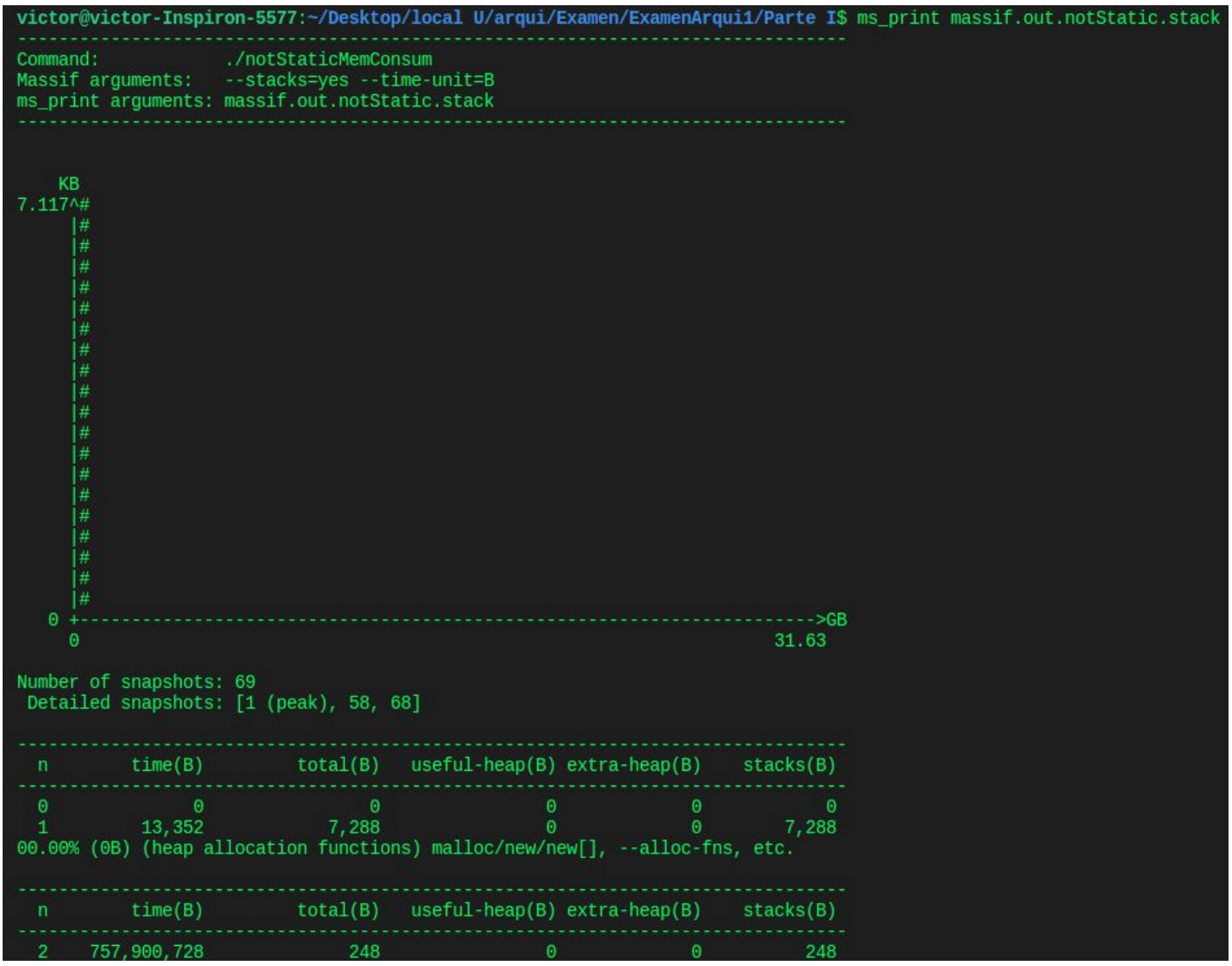

Figura 1.6.3. Salida de análisis del stack con Massif para la función sin variable estática

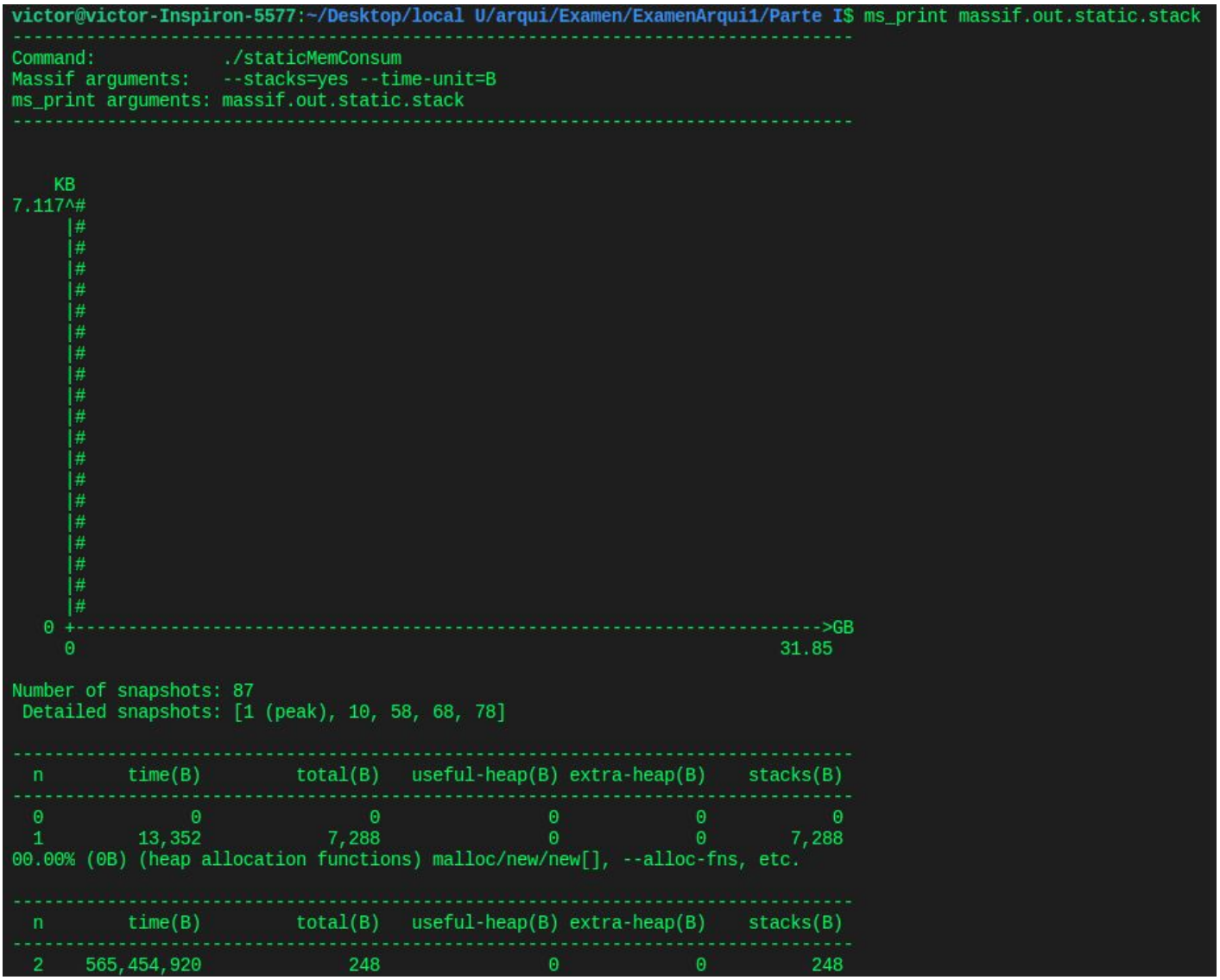

Figura 1.6.4. Salida de análisis del stack con Massif para la función con variable estática

En la figura 1.6.3 se puede observar que el uso de memoria en el punto máximo (indicado por la columna con símbolo #) fue de 7.117KB en el caso de la función sin variable estática y en la figura 1.6.4 se observa el mismo dato en el máximo, por lo que se puede concluir que el uso de la variable estática no afecta el consumo de memoria cuando la tarea realizada es muy simple.

[1.6.1] http://valgrind.org/docs/manual/ms-manual.html

Parte 2

Punto 1

Vector	List
Se almacena en espacios continuos de memoria (implementado como arreglo)	Se almacenan en espacios no contiguos (implementado como lista doblemente enlazada)
Asigna memoria de forma previa para elementos futuros.	No se asigna memoria previa. Sobrecarga en la memoria es constante.
Cada elemento requiere solo el espacio del elemento en sí, no requiere punteros extra.	Por su implementación requiere espacio adicional para la estructura de nodo.
Puede re-asignar memoria para todo el vector al introducir nuevos elementos.	No requiere reajustar el espacio asignado en memoria para sus elementos
Insertar o borrar al final es una operación constante, insertar en una posición específica es O(n)	Insertar y borrar es barato sin importar la posición.
Se puede acceder elementos de forma aleatoria	Acceder un elemento de forma aleatoria es una operación costosa.
Iteradores se invalidan cuando se agregan o remueven elementos.	Iteradores se mantienen válidos.
Se puede extraer fácilmente el arreglo que utilizan por debajo.	Para obtener un arreglo se debe crear este, elemento por elemento.

Tabla 2.1.1 Comparación entre listas y vectores[2.1.1]

[2.1.1]https://stackoverflow.com/questions/2209224/vector-vs-list-in-stl

Punto 2

Un iterador es un objeto, similar a un puntero, que apunta a elementos dentro de un contenedor y mientras un puntero es un ejemplo de iterador, todos los punteros carecen de funcionalidades similares a los punteros. Se pueden clasificar en 5 categorías:

1. **Iteradores de entrada:** Tienen una función muy limitada, solo se pueden usar en algoritmos de una pasada, acceden a archivos del contenedor de forma secuencial de forma que no se repite ningún elemento
2. **Iteradores de salida:** Muy similares a los de entrada pero pero no se usan para acceder datos, sino para asignarles elementos
3. **Iterador hacia adelante:** Contienen todas las funciones de los iteradores de entrada y salida
4. **Iterador bidireccional:** Es un tipo de iterador hacia adelante, que también se puede mover hacia atrás, por eso se llama bidireccional.

5. **Iterador de acceso aleatorio:** Los iteradores más poderosos, no están limitados a moverse de forma secuencial, por lo que pueden acceder a cualquier elemento en el container. Su funcionalidad es la misma que la de un puntero

No todos los containers soportan todos los vectores, para el caso que nos importa, los vectores soportan hasta iteradores de acceso aleatorio mientras que las listas soportan solo hasta iteradores bidireccionales.

ITERATORS	PROPERTIES				
	ACCESS	READ	WRITE	ITERATE	COMPARE
Input	->	= *i		++	==, !=
Output			*i=	++	
Forward	->	= *i	*i=	++	==, !=
Bidirectional		= *i	*i=	++, --	==, !=,
Random-Access	->,[ ]	= *i	*i=	++, --, +=, -==, + ,-	==, !=, <,>,<=,>=

Figura 2.2.1. Operaciones soportadas por contenedores

[2.2.1] https://www.geeksforgeeks.org/introduction-iterators-c/

Punto 3

Se utiliza la herramienta cachegrind de valgrind para llevar a cabo el análisis de caché. Se utilizan banderas para compilar sin optimización y para debuguear, esta última es recomendada por la documentación de cachegrind, mientras que la primera se escoge para asegurar que el programa no sea modificado por el compilador. Para la optimización la documentación solo recomienda correr el programa como normalmente se correría.

Se crean contenedores con 2^{17} valores para la prueba pequeña ya que a partir de 2^{18} cachegrind genera un warning. Por otra parte, se llevan a cabo pruebas con valores mucho más grande, de $2^{25.}$ Aún tras aumentar el *brk_segment* de 8 a 64 MB la advertencia persiste, pero todo indica que el programa se analiza correctamente a pesar de esta. Con esta cantidad de datos la computadora amenaza con congelarse, aunque se puede empujar un poco más (tal vez hasta 2^{27}) De todas formas, el análisis de cachegrind toma alrededor de 350 segundos y aumentar el exponente es un cambio relevante en el tiempo de ejecución.

A continuación se muestran los screenshots con todos los datos disponibles y tablas con los datos de ambos screenshots resumidos.

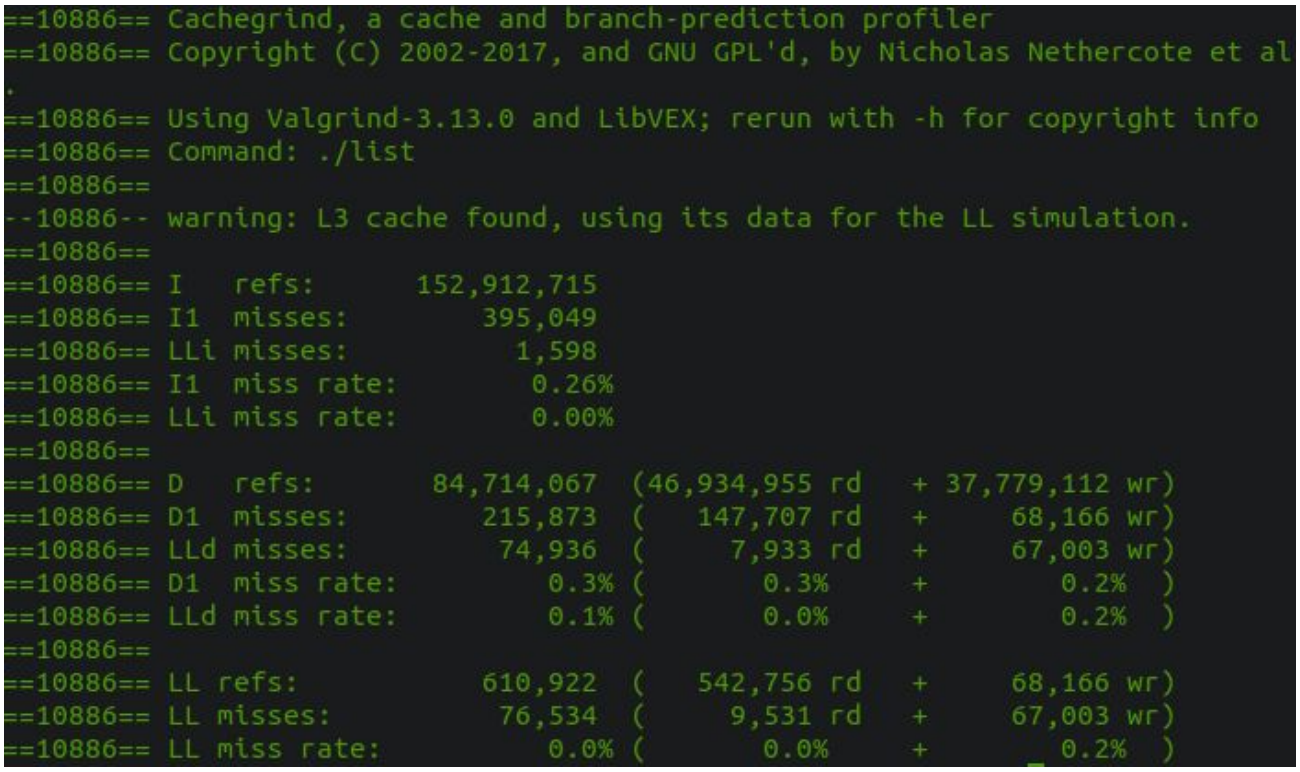

Figura 2.3.1. Uso del caché para la el contenedor lista pequeño

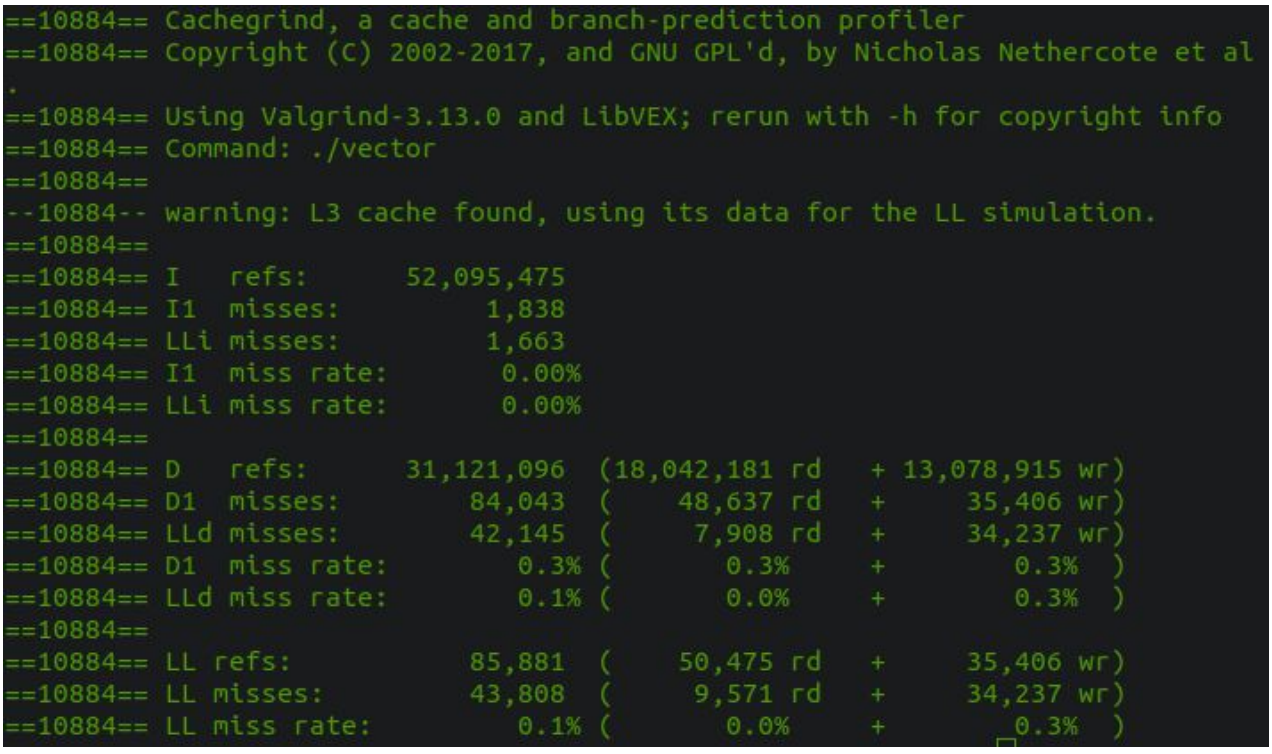

Figura 2.3.2 uso del caché para el contenedor vector pequeño

	Lista (totales)	Vector (totales)
Accesos a instrucciones	152 912 715	52 095 475
Caché L1 de instrucciones - Misses	395 049	1 838
Caché L3 de instrucciones - Misses	1 598	1 663
Caché L1 de instrucciones - Miss Rate	0,26 %	0,00 %
Caché L3 de instrucciones - Miss Rate	0,00 %	0,00 %
Accesos a datos	84 714 067	31 121 096
Caché L1 de datos - Misses	215 873	84 043
Caché L3 de datos - Misses	74 936	42 145
Caché L1 de datos - Miss Rate	0,3 %	0,3 %
Caché L3 de datos - Miss Rate	0,1 %	0,1 %
Accesos a cache de último nivel (L3)	610 922	85 881
Caché de último nivel - Misses	76 534	43 808
Caché de último nivel - Miss Rate	0,0 %	0,1 %

Tabla 2.3.2. Resumen de valores obtenidos en la prueba pequeña

```
==3843== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==3843== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==3843== Command: ./list
==3843==
--3843-- warning: L3 cache found, using its data for the LL simulation.
==3843== brk segment overflow in thread #1: can't grow to 0x4a49000
==3843== (see section Limitations in user manual)
==3843== NOTE: further instances of this message will not be shown
==3843==
==3843== I   refs:      38,587,867,682
==3843== I1  misses:       100,667,140
==3843== LLi misses:             1,626
==3843== I1  miss rate:           0.26%
==3843== LLi miss rate:           0.00%
==3843==
==3843== D   refs:      21,509,173,923  (11,878,852,171 rd   + 9,630,321,752 wr)
==3843== D1  misses:        50,365,046  (    33,580,541 rd   +    16,784,505 wr)
==3843== LLd misses:        50,342,943  (    33,563,695 rd   +    16,779,248 wr)
==3843== D1  miss rate:            0.2% (           0.3%     +           0.2%  )
==3843== LLd miss rate:            0.2% (           0.3%     +           0.2%  )
==3843==
==3843== LL refs:          151,032,186  (   134,247,681 rd   +    16,784,505 wr)
==3843== LL misses:         50,344,569  (    33,565,321 rd   +    16,779,248 wr)
```

Figura 2.3.3. Uso del caché para la el contenedor lista grande

```
812== Cachegrind, a cache and branch-prediction profiler
812== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
812== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
812== Command: ./vector
812==
812-- warning: L3 cache found, using its data for the LL simulation.
812==
812== I   refs:      12,779,843,878
812== I1  misses:             1,913
812== LLi misses:             1,738
812== I1  miss rate:           0.00%
812== LLi miss rate:           0.00%
812==
812== D   refs:       7,785,347,765  (4,480,064,634 rd   + 3,305,283,131 wr)
812== D1  misses:        16,795,259  (    8,403,913 rd   +     8,391,346 wr)
812== LLd misses:        16,328,584  (    7,938,411 rd   +     8,390,173 wr)
812== D1  miss rate:            0.2% (          0.2%     +           0.3%  )
812== LLd miss rate:            0.2% (          0.2%     +           0.3%  )
812==
812== LL refs:           16,797,172  (    8,405,826 rd   +     8,391,346 wr)
812== LL misses:         16,330,322  (    7,940,149 rd   +     8,390,173 wr)
812== LL miss rate:             0.1% (          0.0%     +           0.3%  )
```

Figura 2.3.4 uso del caché para el contenedor vector grande

	Lista (totales)	Vector (totales)
Accesos a instrucciones	38 587 867 682	12 779 843 878
Caché L1 de instrucciones - Misses	100 667 140	1 913
Caché L3 de instrucciones - Misses	1 626	1 738
Caché L1 de instrucciones - Miss Rate	0,26 %	0,00 %
Caché L3 de instrucciones - Miss Rate	0,00 %	0,00 %
Accesos a datos	21 509 173 923	7 785 347 765
Caché L1 de datos - Misses	50 365 046	16 795 259
Caché L3 de datos - Misses	50 342 943	16 328 584
Caché L1 de datos - Miss Rate	0,2 %	0,2 %
Caché L3 de datos - Miss Rate	0,2 %	0,2 %
Accesos a cache de último nivel (L3)	151 032 186	16 797 172
Caché de último nivel - Misses	50 344 569	16 330 322
Caché de último nivel - Miss Rate	0,1 %	0,1 %

Tabla 2.3.2. Resumen de valores obtenidos en la prueba grande

Punto 4

A continuación se presenta una tabla obtenida al dividir los valores obtenidos en la prueba del vector entre los valores obtenidos en la prueba con listas.

	Prueba grande (2^{25})	**Prueba pequeña (2^{17})**
Accesos a instrucciones	33.12%	34.07%
Caché L1 de instrucciones - Misses	0.00%	0.47%
Caché L3 de instrucciones - Misses	106.89%	104.07%
Accesos a datos	36.20%	36.74%
Caché L1 de datos - Misses	33.35%	38.93%
Caché L3 de datos - Misses	32.43%	56.24%
Accesos a cache de último nivel (L3)	11.12%	14.06%
Caché de último nivel - Misses	32.44%	57.24%

Tabla 2.4.1 Comparación entre vector y lista para ambas pruebas

Analizando los datos de las figuras 2.3 por medio de la tabla 2.4.1 en el punto anterior podemos llevar a cabo un análisis. Claramente el uso del caché del programa que utiliza el vector es considerablemente menor, solo un 34% de accesos a instrucciones y un 37% del número de accesos a datos. Los valores permanecen en el rango para la prueba grande, con valores del 33% y 36% a pesar de tener 2 órdenes de magnitud más. Estos resultados son correspondientes con la forma en que se almacena cada uno de estos tipos de datos, ya que la lista se encuentra en espacios no contiguos de memoria, acceder a elementos contiguos en la lista puede implicar acceder 3 posiciones de memoria diferentes, en vez de solo 1 posición (y jalar los datos alrededor de esta) y vemos cómo esta razón de ⅓ se ve representada en los accesos que se aproximan a 33%.

De forma similar en el último nivel de caché, que valgrind determinar cómo el nivel L3, ve un decremento al 14% de accesos al utilizar vectores, en vez de listas y este cambio se todavía más en la prueba grande que decrementa al 11% esto se puede dar porque es el tercer nivel de caché y solo recibe búsquedas de elementos que no están en los primeros dos niveles. Como vemos una gran discrepancia entre los accesos en el primer nivel contra los del tercer nivel podemos ver que muchos de los accesos quedan en el nivel 2 del caché.

La cantidad de misses aumenta junto con el tamaño de la prueba lo que es de esperar ya que hay más accesos a memoria pero, de forma interesante el miss rate se mantiene bastante constante incluso cuando la cantidad de llamas se reduce, en la prueba pequeña. En la prueba grande parece que los datos que no se encontraron en L1 no estaban del todo en caché y resultaron en prácticamente la misma cantidad de misses en L3.

AMD FX-8350	Tamaño
Total L1 Cache	384 KB
Total L2 Cache	8 MB
Total L3 Cache	8 MB

Tabla 2.4.2 Características del sistema

El miss rate de 0% en el cache L3, nos indica que mientras el programa de lista no se puede reservar en los 384 KB que tiene el sistema en el primer nivel, el programa sí se encuentra en alguno de los niveles inferiores de caché, por otro lado, como el programa con vector tiene un miss rate del 0% en L1, parece tener un memory footprint menor y sí se puede reservar casi todo en el caché de primer nivel.

No espera ver resultados relevantes respecto a la inserción de elementos en los contenedores ya que ambos permiten el uso de push_back para agregar elementos al final del contenedor y así fue implementado para ambos casos, se asume que el costo de inserción al final es muy parecido ya que en las diferencias que encontramos solo se hace referencia a un mayor costo al insertar un elemento en el medio del vector y la asignación de memoria para el vector sólo se hace una vez, mientras que la de lista se hace al azar, por lo que a pesar de que se hace más veces, se espera que tenga un bajo costo en ejecución.