|  |  |
| --- | --- |
| **Assignment** | |
| **Course Code** | CSC305A |
| **Course Name** | Programming Language Principles |
| **Programme** | B.Tech |
| **Department** | CSE |
| **Faculty** | FET |

|  |  |
| --- | --- |
| **Name of the Student** | Satyajit Ghana |
| **Reg. No.** | 17ETCS002159 |
| **Semester/Year** | 05/2019 |
| **Course Leader(s)** | Ms. Chaitra S |



|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Declaration Sheet | | | | | | | | |
| Student Name | Satyajit Ghana | | | | | | | |
| Reg. No | 17ETCS002159 | | | | | | | |
| Programme | B.Tech | | | | | Semester/Year | 05/2019 | |
| Course Code | CSC305A | | | | | | | |
| Course Title | Programming Language Principles | | | | | | | |
| Course Date |  | | to |  | | | | |
| Course Leader | Ms. Chaitra S | | | | | | | |
| **Declaration**  The assignment submitted herewith is a result of my own investigations and that I have conformed to the guidelines against plagiarism as laid out in the Student Handbook. All sections of the text and results, which have been obtained from other sources, are fully referenced. I understand that cheating and plagiarism constitute a breach of University regulations and will be dealt with accordingly. | | | | | | | | |
| Signature of the Student | |  | | | | | Date |  |
| Submission date stamp  (by Examination & Assessment Section) | |  | | | | | | |
| Signature of the Course Leader and date | | | | | Signature of the Reviewer and date | | | |
|  | | | | |  | | | |

# Contents

[Declaration Sheet ii](#_Toc24316558)

[Contents iii](#_Toc24316559)

[List of Tables iv](#_Toc24316560)

[1 Question 1 5](#_Toc24316561)

[1.1 Implementation of the application using C-Programming 5](#_Toc24316562)

[1.2 Implementation of the application using JAVA-Programming using OOP features 9](#_Toc24316563)

[1.3 Discussion on ease of writing program in C in comparison with that in JAVA 10](#_Toc24316564)

[1.4 Discussion on the amount of changes required to introduce a new tax slab 25% and removing 28% slab in both the languages 11](#_Toc24316565)

[1.5 Discussion on the efficiency in terms of CPU and MEM usage using tools in both the languages 11](#_Toc24316566)

[1.5.1 Performance results analysis 15](#_Toc24316567)

[Appendix A 17](#_Toc24316568)

[Appendix B 19](#_Toc24316569)

[Bibliography 23](#_Toc24316570)

# List of Tables

[Table 1 Performance Comparison 15](#_Toc21744065)

[Table 2 Memory Comparison 15](#_Toc21744066)

# Question 1

Solution to Question No. 1

## Implementation of the application using C-Programming

**main.c**

#include <stdio.h>

#include "array\_list.h"

#include "gst\_item.h"

#define MAX\_ITEMS 100000

**int** main(**int** argc, **char** \*\*argv) {

*// testing*

    array\_list \*list = init\_array\_list(free\_gst\_item);

*/\*\**

*\* A Single loop of 100000 objects, sequencial access*

*\*/*

    for (**int** i = 0; i < MAX\_ITEMS; i++) {

**int** item\_id = i;

**int** gst\_slab\_id = i % 5;

**double** price = 1000.5f;

        gst\_item \*item = create\_gst\_item(item\_id, gst\_slab\_id, price);

        list->add(list, item);

    }

**double** total\_amt = 0.0f;

*// calculate the GST'ed price*

    for (**int** i = 0; i < MAX\_ITEMS; i++) {

        gst\_item \*item = list->at(list, i);

        total\_amt += get\_mrp(item);

    }

    free\_array\_list(list);

    printf("TOTAL\_ITEMS : %d\nTOTAL AMOUNT : %.10f\n", MAX\_ITEMS, total\_amt);

    fflush(stdout);

    return 0;

}

**array\_list.h**

#pragma once

**struct** array\_list {

**void**\*\* data;

**void\*** (\*at)(**struct** array\_list**\***, **int** idx);

**int** (\*add)(**struct** array\_list**\***, **void\*** data);

**void** (\*free\_item)(**void\*** data);

**int** length;

**int** buffered\_length;

};

typedef **struct** array\_list array\_list;

array\_list**\*** init\_array\_list(**void** (\*)(**void\***));

**void\*** array\_list\_at(**struct** array\_list**\***, **int** idx);

**int** array\_list\_add(**struct** array\_list**\***, **void\*** data);

**void** free\_array\_list\_items(array\_list**\***);

**void** free\_array\_list(array\_list**\***);

**array\_list.c**

#include "array\_list.h"

#include <stdio.h>

#include <stdlib.h>

*// set the buffer to 256 elements*

#define BUFFER 256

array\_list\* init\_array\_list(**void** (\*free\_item)(**void**\*)) {

    array\_list\* mylist;

    if ((mylist = malloc(sizeof \*mylist)) != NULL) {

        mylist->at = array\_list\_at;

        mylist->add = array\_list\_add;

        mylist->buffered\_length = BUFFER;

        mylist->free\_item = free\_item;

        mylist->data = malloc(mylist->buffered\_length \* sizeof(\*mylist->data));

        if (mylist->data == NULL) {

            return NULL;

        }

        return mylist;

    }

    return NULL;

}

**void**\* array\_list\_at(**struct** array\_list\* mylist, **int** idx) {

    if (mylist != NULL) {

        if (idx >= mylist->length) {

            perror("idx out of bounds\n");

            return NULL;

        }

        return mylist->data[idx];

    }

    return NULL;

}

**int** array\_list\_add(**struct** array\_list\* mylist, **void**\* data) {

    if (mylist != NULL) {

        if (mylist->length >= mylist->buffered\_length) {

            mylist->buffered\_length += BUFFER;

            mylist->data = realloc(mylist->data, (mylist->buffered\_length) \* sizeof(\*mylist->data));

        }

        mylist->data[mylist->length++] = data;

        return 1;

    }

    return -1;

}

**void** free\_array\_list\_items(array\_list\* list) {

    for (**int** i = 0; i < list->length; i++) {

        list->free\_item(list->at(list, i));

    }

    list->length = 0;

    list->buffered\_length = 0;

}

**void** free\_array\_list(array\_list\* list) {

    free\_array\_list\_items(list);

    free(list);

}

**gst\_item.h**

#pragma once

**struct** gst\_item {

**int** item\_id;

**int** gst\_slab\_id;

**double** price;

};

typedef **struct** gst\_item gst\_item;

**static** **float** slab\_tax[5] = {0.0, 0.05, 0.12, 0.18, 0.28};

**float** get\_slab\_tax(**int** slab\_id);

**double** get\_mrp(gst\_item**\*** item);

**void\*** create\_gst\_item(**int**, **int**, **double**);

**void** free\_gst\_item(**void\***);

**gst\_item.c**

#include "gst\_item.h"

#include <assert.h>

#include <stdio.h>

#include <stdlib.h>

**float** get\_slab\_tax(**int** slab\_id) {

    return slab\_tax[slab\_id];

}

**double** get\_mrp(gst\_item\* item) {

    assert(item != NULL);

    return item->price + get\_slab\_tax(item->gst\_slab\_id) \* item->price;

}

**void**\* create\_gst\_item(**int** item\_id, **int** slab\_id, **double** price) {

    gst\_item\* item;

    if ((item = malloc(sizeof \*item)) != NULL) {

        item->item\_id = item\_id;

        item->gst\_slab\_id = slab\_id;

        item->price = price;

        return item;

    }

    return NULL;

}

**void** free\_gst\_item(**void**\* item) {

    gst\_item\* myitem = item;

    free(myitem);

}

## Implementation of the application using JAVA-Programming using OOP features

**App.java**

package **app**;

import **java.util.ArrayList**;

**public** **class** App {

**public** **static** **int** MAX\_ITEMS = 100000;

**public** **static** **void** main(**String**[] args) **throws** **Exception** {

*// testing*

**ArrayList**<**GSTItem**> list = new **ArrayList**<>();

*/\*\**

*\* A Single loop of 100000 objects, sequential access*

*\*/*

        for (**int** i = 0; i < MAX\_ITEMS; i++) {

**int** item\_id = i;

**int** gst\_slab\_id = i % 5;

**double** price = 1000.5f;

**GSTItem** item = new GSTItem(item\_id, gst\_slab\_id, price);

            list.add(item);

        }

**double** total\_amt = 0.0f;

*// calculate the GST'ed price*

        for (**int** i = 0; i < MAX\_ITEMS; i++) {

**GSTItem** item = list.get(i);

            total\_amt += item.get\_mrp();

        }

        System.out.printf("TOTAL\_ITEMS : %d\nTOTAL\_AMOUNT : %.10f\n", MAX\_ITEMS, total\_amt);

    }

}

**GSTItem.java**

package **app**;

*/\*\**

*\* GSTItem*

*\*/*

**public** **class** GSTItem {

**private** **int** item\_id;

**private** **int** gst\_slab\_id;

**private** **double** price;

    GSTItem(**int** item\_id, **int** gst\_slab\_id, **double** price) {

        this.item\_id = item\_id;

        this.gst\_slab\_id = gst\_slab\_id;

        this.price = price;

    }

**public** **static** **float** slab\_tax[] = { 0.0f, 0.05f, 0.12f, 0.18f, 0.28f };

**public** **double** get\_mrp() {

        return this.price + slab\_tax[this.gst\_slab\_id] \* this.price;

    }

}

## Discussion on ease of writing program in C in comparison with that in JAVA

C is a bare bones language, i.e. it is much more primitive and has very low-level access to hardware, the inbuilt libraries that come with C is limited and most of the memory allocation is done manually by the programmer, this gives more control over the memory used by the program.

When it comes to Java there are a lot of inbuilt libraries that come with it, which makes it easier to write code, the memory management is done by JVM and deletion of variables is not given to the programmer.

For example, in the program that was implemented in 1.2 and 1.3, a data structure was required to store the GST Items, such as an ArrayList, by default an arraylist is not available in C, an array is available, but it is not generic, i.e. the array cannot change the data that it holds at runtime, therefore we implemented a custom array\_list Abstract Data Type that can store any kind of data, now the memory management and the functions associated with this ADT was written manually by the programmer, which takes a lot of effort and the programmer needs to be very careful about the memory leaks that might happen if the ADT is not disposed properly. When it came to JAVA, we already have a proper implementation of ArrayList which can be directly imported from the library and used, all the basic functions are already defined, hence our program source code is much more smaller than C.

## Discussion on the amount of changes required to introduce a new tax slab 25% and removing 28% slab in both the languages

The slab tax is stored in form of an array in both the languages, since the slabs are already predefined and fixed. Now if we have to introduce a new tax and remove the existing one, the number can be simply changed in the array. The new lines of code that will make the necessary changes:

In C

static float slab\_tax[5] = {0.0, 0.05, 0.12, 0.18, 0.25};

In JAVA

public static float slab\_tax[] = { 0.0f, 0.05f, 0.12f, 0.18f, 0.25f };

## Discussion on the efficiency in terms of CPU and MEM usage using tools in both the languages

Tests were conducted using two tools, common for both Java and C. The testing was done on a Linux Machine, using Bash Scripting to automate the process, the bash script is attached in Appendix A.

Memory Profiling: memusage

CPU Profiling: perf

OS: Arch Linux Kernel 4.9

CPU: Intel Core [i3-6006U@2.93Ghz](mailto:i3-6006U@2.93Ghz) (2 Cores, 4 HyperThreads)

Below is the RAW statistics provided by these tools

**Language: C**

**CPU PERFORMANCE (PERF)**

TOTAL\_ITEMS : 100000

TOTAL AMOUNT : 112656300.1282616258

Performance counter stats for 'gst-c/build/gst-c':

9.54 msec task-clock:u # 0.967 CPUs utilized

0 context-switches:u # 0.000 K/sec

0 cpu-migrations:u # 0.000 K/sec

1,031 page-faults:u # 0.108 M/sec

13,126,242 cycles:u # 1.376 GHz

46,010,443 instructions:u # 3.51 insn per cycle

9,413,655 branches:u # 986.741 M/sec

4,664 branch-misses:u # 0.05% of all branches

0.009869029 seconds time elapsed

0.003299000 seconds user

0.006539000 seconds sys

# Overhead Command Shared Object Symbol

34.66% gst-c libc-2.29.so [.] \_int\_malloc

12.91% gst-c libc-2.29.so [.] \_int\_free

10.70% gst-c [unknown] [k] 0xffffffff8a800b07

10.13% gst-c libc-2.29.so [.] malloc

6.74% gst-c libc-2.29.so [.] malloc\_consolidate

6.33% gst-c libc-2.29.so [.] cfree@GLIBC\_2.2.5

5.28% gst-c gst-c [.] main

4.76% gst-c gst-c [.] get\_mrp

4.14% gst-c gst-c [.] create\_gst\_item

2.87% gst-c gst-c [.] array\_list\_at

1.03% gst-c ld-2.29.so [.] strcmp

0.46% gst-c ld-2.29.so [.] \_\_GI\_\_\_tunables\_init

**MEMORY USAGE (MEMUSAGE)**

Memory usage summary: heap total: 2404904, heap peak: 2400808, stack peak: 2112

total calls total memory failed calls

malloc| 100003 1606184 0

realloc| 390 798720 0 (nomove:342, dec:0, free:0)

calloc| 0 0 0

free| 100001 1600040

Histogram for block sizes:

16-31 100000 99% ==================================================

32-47 1 <1%

2048-2063 1 <1%

4096-4111 2 <1%

6144-6159 1 <1%

8192-8207 1 <1%

10240-10255 1 <1%

12288-12303 1 <1%

14336-14351 1 <1%

16384-16399 1 <1%

18432-18447 1 <1%

20480-20495 1 <1%

22528-22543 1 <1%

24576-24591 1 <1%

26624-26639 1 <1%

28672-28687 1 <1%

30720-30735 1 <1%

32768-32783 1 <1%

34816-34831 1 <1%

36864-36879 1 <1%

38912-38927 1 <1%

40960-40975 1 <1%

43008-43023 1 <1%

45056-45071 1 <1%

47104-47119 1 <1%

49152-49167 1 <1%

51200-51215 1 <1%

53248-53263 1 <1%

55296-55311 1 <1%

57344-57359 1 <1%

59392-59407 1 <1%

61440-61455 1 <1%

63488-63503 1 <1%

large 360 <1%

**Language: JAVA**

**CPU PERFORMANCE (PERF)**

TOTAL\_ITEMS : 100000

TOTAL\_AMOUNT : 112656300.1282616300

Performance counter stats for 'java -cp GSTJava/bin app.App':

212.06 msec task-clock:u # 1.459 CPUs utilized

0 context-switches:u # 0.000 K/sec

0 cpu-migrations:u # 0.000 K/sec

4,752 page-faults:u # 0.022 M/sec

359,432,031 cycles:u # 1.695 GHz

411,960,222 instructions:u # 1.15 insn per cycle

78,327,254 branches:u # 369.371 M/sec

2,874,409 branch-misses:u # 3.67% of all branches

0.145300220 seconds time elapsed

0.189254000 seconds user

0.026494000 seconds sys

# Overhead Command Shared Object Symbol

2.14% java libjvm.so [.] 0x0000000000d14785

1.14% java libjvm.so [.] 0x0000000000d147b1

0.82% java [unknown] [k] 0xffffffff8a800b07

0.78% java libjvm.so [.] 0x0000000000b7a168

0.63% java libjvm.so [.] 0x0000000000b7a5fb

0.63% java libjvm.so [.] 0x0000000000d147ae

0.54% C1 CompilerThre ld-2.29.so [.] \_\_tls\_get\_addr

0.52% java libjvm.so [.] 0x0000000000d147b7

0.52% java [JIT] tid 1347 [.] 0x00007f9ab89654c4

0.47% java libjvm.so [.] 0x0000000000d14b69

0.44% java libc-2.29.so [.] \_\_vfprintf\_internal

0.43% java ld-2.29.so [.] do\_lookup\_x

0.43% java [JIT] tid 1347 [.] 0x00007f9ab8957096

0.39% java libjimage.so [.] 0x0000000000002d15

0.39% java [JIT] tid 1347 [.] 0x00007f9ab89780fb

0.39% java libjvm.so [.] 0x0000000000c6a4e0

0.38% java [JIT] tid 1347 [.] 0x00007f9ab89654c0

0.38% java [JIT] tid 1347 [.] 0x00007f9ab8974702

0.38% java libjvm.so [.] 0x0000000000bb074a

0.38% java libjvm.so [.] 0x00000000008246cc

0.38% java [JIT] tid 1347 [.] 0x00007f9ab8965516

0.37% java libjimage.so [.] 0x0000000000003a1f

0.37% java ld-2.29.so [.] \_dl\_relocate\_object

0.37% java libjvm.so [.] 0x00000000008246c7

0.36% C1 CompilerThre libc-2.29.so [.] \_\_vfscanf\_internal

0.36% java [JIT] tid 1347 [.] 0x00007f9ab896bd88

0.35% java libc-2.29.so [.] \_int\_malloc

0.30% C1 CompilerThre libc-2.29.so [.] \_\_memmove\_avx\_unaligned\_erms

0.29% java ld-2.29.so [.] strcmp

0.27% C1 CompilerThre [JIT] tid 1347 [.] 0x00007f9ab8957096

0.27% java [JIT] tid 1347 [.] 0x00007f9ab8969568

0.26% C1 CompilerThre libc-2.29.so [.] [cfree@GLIBC\_2.2.5](mailto:cfree@GLIBC_2.2.5)

**MEMORY USAGE (MEMUSAGE)**

Memory usage summary: heap total: 14905189, heap peak: 12809891, stack peak: 30688

total calls total memory failed calls

malloc| 11386 14668887 0

realloc| 38 1120 0 (nomove:0, dec:0, free:0)

calloc| 110 235182 0

free| 4545 2169338

Histogram for block sizes:

0-15 383 3% ===

16-31 5356 46% ==================================================

32-47 2258 19% =====================

48-63 1224 10% ===========

64-79 375 3% ===

80-95 153 1% =

96-111 56 <1%

112-127 167 1% =

128-143 111 <1% =

144-159 22 <1%

160-175 27 <1%

176-191 64 <1%

192-207 18 <1%

208-223 10 <1%

224-239 37 <1%

240-255 54 <1%

256-271 25 <1%

272-287 53 <1%

288-303 68 <1%

304-319 28 <1%

320-335 18 <1%

336-351 128 1% =

352-367 2 <1%

368-383 62 <1%

384-399 9 <1%

400-415 2 <1%

432-447 65 <1%

464-479 2 <1%

480-495 2 <1%

496-511 2 <1%

512-527 3 <1%

528-543 2 <1%

544-559 147 1% =

### Performance results analysis

Both the programs were run for the exact same data for the exact same number of iterations, i.e. 100000 items with varying types of GST Tax slabs and the MRP for each of these commodities is calculated by the program. The program was again run for 10000000 items again and the results have been attached in Appendix B.

Here is a table that describes the summary of the performance parameters

Table Performance Comparison

|  |  |  |
| --- | --- | --- |
| **Parameter** | **C** | **JAVA** |
| Execution Time | 0.009869s | 0.145300s |
| IPC | 3.51 ins/cycle | 1.15 ins/cycle |
| Instructions | 46,010,443 ins | 411,960,222 ins |

We can clearly observe in this scenario C is approximately 15 times faster than Java, this is without the optimizations done in C, which will make it even faster. Since Java is an interpreted language it is bound to be slower, the byte code generated is run on a JVM that uses the JIT Compiler to compile and run the code.

Here is a table that compares the memory usage of the two languages

Table Memory Comparison

|  |  |  |
| --- | --- | --- |
| **Parameter** | **C** | **JAVA** |
| malloc | 1568.53 kB | 14325.08 kB |
| realloc | 780 kB | 1.09 kB |
| calloc | 0 kB | 229.67 kB |
| free | 1562.53 kB | 2118.49 kB |
| heap total | 2348.53 kB | 14555.84 kB |
| stack peak | 2.06 kB | 29.96 kB |

Similarly, as observed previously, C is approximately 7 times less memory consuming, comparing the malloc and stack peak. malloc is the main routine in Linux libc that allocates memory for the program at runtime, both our programs make runtime memory allocation to make this comparison fair. Another thing to note is that the amount of memory allocated by malloc and then freed is important, since this shows the memory leaks that can happen in the program.

Another interesting thing to note in the memusage dump is that C allocated 99% of its memory chunks of the size 15-31 bytes, i.e. most of the memory that is used by the program was only for the gst item objects that we allocated at runtime. While in Java the allocation is distributed only about 46% of its memory was 15-31 byte chunks, this is because of the other objects that are associated with the gst item that also need to be allocated along with it.

In C the difference of malloc and free is 6kB while in JAVA it is 12206kB, which means that this memory wasn’t freed at runtime, it must have been freed either by the JVM or by the OS when the program terminated.

The conclusion from this analysis is that JAVA has much more overhead than C since it is interpreted and has many inbuilt complex data structures that require a lot of runtime overhead, this contributes to the slower runtime and the higher memory usage. This can be seen in the Overhead displayed in perf analysis, Java several calls to [JIT] tid <tid>.

# Appendix A

*#!/bin/bash*

COLS=$(tput cols)

function print\_head {

    print\_line

    echo -e "$1\n"

}

function print\_line {

    printf '=%.0s' $(seq $COLS)

    printf "\n"

}

print\_head ""

start=$SECONDS

echo -e "Bulding the Java Byte Code\n"

(cd GSTJava**;** javac -verbose -cp src src/app/App.java -d bin/)

echo -e "\nBuilding the C executable Code\n"

cd gst-c

rm -r build **&>** /dev/null

mkdir build

(cd build**;** cmake -DCMAKE\_BUILD\_TYPE=Release ..**;** make)

cd ..

rm -r mem\_report **&>** /dev/null

mkdir mem\_report

echo -e "Testing Memory Usage for Java\n"

memusage java -cp GSTJava/bin app.App **|&** sed -r "s/\x1B\[([0-9]{1,2}(;[[0-9]{1,2})?)[mGK]//g" **|&** tee mem\_report/java\_mem.txt

echo -e "Testing Memory Usage for C\n"

memusage gst-c/build/gst-c **|&** sed -r "s/\x1B\[([0-9]{1,2}(;[[0-9]{1,2})?)[mGK]//g" **|&** tee mem\_report/c\_mem.txt

*# Perf Report Directory*

rm -r perf\_report **&>** /dev/null

mkdir perf\_report

echo -e "Testing Performance Stats for Java\n"

perf stat java -cp GSTJava/bin app.App **|&** tee perf\_report/java\_perf\_stat.txt

echo -e "Testing Performance Stats for C\n"

perf stat gst-c/build/gst-c **|&** tee perf\_report/c\_perf\_stat.txt

echo -e "Running perf record and saving the output (could take a while ..)\n"

perf record -s java -cp GSTJava/bin app.App

mv perf.data perf\_report/java\_perf.data

perf report --stdio -i perf\_report/java\_perf.data **|&** tee perf\_report/java\_perf\_report.txt

perf record -s gst-c/build/gst-c

mv perf.data perf\_report/c\_perf.data

perf report --stdio -i perf\_report/c\_perf.data **|&** tee perf\_report/c\_perf\_report.txt

echo -e "Perf records saved in perf\_report/\n"

rm -r **\***.data **&>** /dev/null

duration=$(( SECONDS**-**start ))

echo -e "testing took $duration seconds\n"

print\_head ""

# Appendix B

**Language: C**

**CPU PERFORMANCE (PERF)**

TOTAL\_ITEMS : 10000000

TOTAL AMOUNT : 11265630013.4887523651

Performance counter stats for 'gst-c/build/gst-c':

976.70 msec task-clock:u # 0.978 CPUs utilized

0 context-switches:u # 0.000 K/sec

0 cpu-migrations:u # 0.000 K/sec

97,709 page-faults:u # 0.100 M/sec

1,334,998,758 cycles:u # 1.367 GHz

4,206,621,972 instructions:u # 3.15 insn per cycle

901,448,622 branches:u # 922.954 M/sec

331,942 branch-misses:u # 0.04% of all branches

0.998173867 seconds time elapsed

0.685913000 seconds user

0.272802000 seconds sys

# Overhead Command Shared Object Symbol

# ........ ....... ................ .......................

#

26.34% gst-c libc-2.30.so [.] \_int\_malloc

16.11% gst-c libc-2.30.so [.] \_int\_free

11.26% gst-c libc-2.30.so [.] malloc\_consolidate

9.82% gst-c libc-2.30.so [.] malloc

8.84% gst-c gst-c [.] main

5.52% gst-c [unknown] [k] 0xffffffff9ea00b07

4.17% gst-c libc-2.30.so [.] cfree@GLIBC\_2.2.5

3.63% gst-c gst-c [.] create\_gst\_item

3.53% gst-c gst-c [.] array\_list\_add

3.46% gst-c gst-c [.] array\_list\_at

2.67% gst-c gst-c [.] get\_mrp

2.12% gst-c gst-c [.] free\_array\_list

0.85% gst-c gst-c [.] free@plt

0.58% gst-c gst-c [.] free\_gst\_item

0.56% gst-c gst-c [.] malloc@plt

0.19% gst-c libc-2.30.so [.] mremap\_chunk

0.12% gst-c libc-2.30.so [.] realloc

0.08% gst-c [unknown] [k] 0xffffffff9ea0015f

0.03% gst-c libc-2.30.so [.] unlink\_chunk.isra.0

**MEMORY USAGE (MEMUSAGE)**

TOTAL\_ITEMS : 10000000

TOTAL AMOUNT : 11265630013.4887523651

Memory usage summary: heap total: 240005160, heap peak: 240001064, stack peak: 2112

total calls total memory failed calls

malloc| 10000003 160006184 0

realloc| 39062 79998976 0 (nomove:39007, dec:0, free:0)

calloc| 0 0 0

free| 10000001 160000040

Histogram for block sizes:

16-31 10000000 99% ==================================================

32-47 1 <1%

2048-2063 1 <1%

4096-4111 2 <1%

6144-6159 1 <1%

8192-8207 1 <1%

10240-10255 1 <1%

12288-12303 1 <1%

14336-14351 1 <1%

16384-16399 1 <1%

18432-18447 1 <1%

20480-20495 1 <1%

22528-22543 1 <1%

24576-24591 1 <1%

26624-26639 1 <1%

28672-28687 1 <1%

30720-30735 1 <1%

32768-32783 1 <1%

34816-34831 1 <1%

36864-36879 1 <1%

38912-38927 1 <1%

40960-40975 1 <1%

43008-43023 1 <1%

45056-45071 1 <1%

47104-47119 1 <1%

49152-49167 1 <1%

51200-51215 1 <1%

53248-53263 1 <1%

55296-55311 1 <1%

57344-57359 1 <1%

59392-59407 1 <1%

61440-61455 1 <1%

63488-63503 1 <1%

large 39032 <1%

**Language: JAVA**

**CPU PERFORMANCE (PERF)**

TOTAL\_ITEMS : 10000000

TOTAL\_AMOUNT : 11265630013.4887520000

Performance counter stats for 'java -cp GSTJava/bin app.App':

4,296.89 msec task-clock:u # 1.582 CPUs utilized

0 context-switches:u # 0.000 K/sec

0 cpu-migrations:u # 0.000 K/sec

146,928 page-faults:u # 0.034 M/sec

7,278,982,030 cycles:u # 1.694 GHz

13,001,805,299 instructions:u # 1.79 insn per cycle

2,081,763,347 branches:u # 484.481 M/sec

4,981,395 branch-misses:u # 0.24% of all branches

2.716024623 seconds time elapsed

3.715426000 seconds user

0.560803000 seconds sys

# Overhead Command Shared Object Symbol

# ........ ............... .................. ...................................................................

#

3.13% java [JIT] tid 917 [.] 0x00007f15a442477f

2.27% G1 Conc#0 libjvm.so [.] 0x000000000074df80

2.02% G1 Conc#0 libjvm.so [.] 0x0000000000747093

1.58% GC Thread#0 libjvm.so [.] 0x0000000000782cab

1.28% GC Thread#1 libjvm.so [.] 0x0000000000782cab

1.21% G1 Conc#0 libjvm.so [.] 0x000000000074dfe3

1.11% java [unknown] [k] 0xffffffff9ea00b07

1.01% G1 Conc#0 libjvm.so [.] 0x000000000080eccc

0.94% GC Thread#2 libjvm.so [.] 0x0000000000782cab

0.82% java libjvm.so [.] 0x0000000000a44144

0.81% GC Thread#0 libjvm.so [.] 0x00000000007852fb

0.81% G1 Conc#0 libjvm.so [.] 0x00000000007470db

0.79% GC Thread#3 libjvm.so [.] 0x0000000000782cab

0.71% GC Thread#1 libjvm.so [.] 0x00000000007852fb

0.64% GC Thread#2 libjvm.so [.] 0x00000000007852fb

0.62% G1 Refine#0 libjvm.so [.] 0x000000000079343c

0.59% G1 Conc#0 libjvm.so [.] 0x000000000079401a

0.59% G1 Refine#0 libjvm.so [.] 0x000000000080eccc

0.55% java [JIT] tid 917 [.] 0x00007f15a44251ea

0.50% GC Thread#0 libjvm.so [.] 0x000000000078571f

0.49% G1 Conc#0 libjvm.so [.] 0x0000000000793c46

0.47% G1 Conc#0 libjvm.so [.] 0x0000000000794022

0.47% G1 Conc#0 libjvm.so [.] 0x0000000000793c79

0.45% G1 Refine#0 libjvm.so [.] 0x0000000000793431

0.44% G1 Conc#0 libjvm.so [.] 0x0000000000793c62

0.44% GC Thread#1 libjvm.so [.] 0x000000000078571f

0.44% G1 Conc#0 libjvm.so [.] 0x0000000000793c83

0.43% G1 Refine#0 libjvm.so [.] 0x000000000080ecc1

0.42% G1 Refine#0 libjvm.so [.] 0x0000000000793435

0.40% GC Thread#2 libjvm.so [.] 0x000000000078571f

0.39% GC Thread#3 libjvm.so [.] 0x00000000007852fb

0.38% G1 Conc#0 libjvm.so [.] 0x000000000079507b

0.38% G1 Conc#0 libjvm.so [.] 0x000000000080eca9

0.37% G1 Conc#0 libjvm.so [.] 0x0000000000793c8e

0.36% G1 Conc#0 libjvm.so [.] 0x000000000080ec9a

0.36% G1 Conc#0 libjvm.so [.] 0x0000000000793c67

0.36% java libjvm.so [.] 0x000000000072e401

0.35% G1 Conc#0 libjvm.so [.] 0x000000000074702c

0.35% java [JIT] tid 917 [.] 0x00007f15a44251f6

0.34% G1 Conc#0 libjvm.so [.] 0x000000000080edb7

0.32% G1 Conc#0 libjvm.so [.] 0x000000000080ecbe

0.32% java [JIT] tid 917 [.] 0x00007f15a4425215

0.32% G1 Conc#0 libjvm.so [.] 0x0000000000793c50

**MEMORY USAGE (MEMUSAGE)**

TOTAL\_ITEMS : 10000000

TOTAL\_AMOUNT : 11265630013.4887520000

Memory usage summary: heap total: 18350583, heap peak: 15513998, stack peak: 26240

total calls total memory failed calls

malloc| 18916 18110689 0

realloc| 62 2768 0 (nomove:2, dec:0, free:0)

calloc| 116 237126 0

free| 4983 2910593

Histogram for block sizes:

0-15 407 2% ===

16-31 5423 28% ==================================================

32-47 2308 12% =====================

48-63 1278 6% ===========

64-79 2924 15% ==========================

80-95 242 1% ==

96-111 56 <1%

112-127 785 4% =======

128-143 111 <1% =

144-159 21 <1%

160-175 65 <1%

176-191 673 3% ======

192-207 19 <1%

208-223 10 <1%

224-239 50 <1%

240-255 62 <1%

256-271 1281 6% ===========

272-287 53 <1%

288-303 676 3% ======

304-319 31 <1%

320-335 72 <1%

336-351 130 <1% =

352-367 6 <1%

368-383 666 3% ======

384-399 9 <1%

400-415 2 <1%

432-447 93 <1%

448-463 10 <1%

464-479 44 <1%

480-495 164 <1% =

496-511 2 <1%

512-527 7 <1%

528-543 7 <1%

544-559 1 <1%

560-575 8 <1%

576-591 4 <1%

624-639 30 <1%

# Bibliography