CMU (Carnegie Mellon University)
Analysis of Software Artifacts

Mini Project

17-654/17-754: Analysis of Software Artifacts

Analysis Application of Rational Purify™: Utilization of Purify in the Navigation Data Converter Application

April 27, 2006

Pathfinder Team

Wangbong Lee (*wangbonl*)
Jihye Eom (*jihyee*)
Youngseok Oh (*youngseo*)

1 Tool Introduction

1.1 Brief introduction

A memory leak is unnecessary memory consumption by a program, where the program fails to release memory that is no longer needed. The memory leak can diminish the performance of the program by reducing the amount of available memory. Memory allocation is normally a component of the operating system, so the result of a memory leak usually an ever growing amount of memory being used by the system as a whole, not merely by the erroneous program. Eventually, too much of the available memory may become allocated and all or part of the system stops working correctly or the application fails. [5].

Memory leaks are a common error in programming when using languages that have no automatic garbage collection, such as C and C++. Typically, a memory leak occurs because dynamically allocated memory has become unreachable. The prevalence of memory leak bugs has led to the development of a number of debugging tools to detect unreachable memory.[5] Therefore, many memory debuggers are used such as Purify[7], Valgrind[8], Insure++[9], and memwatch[10].

In addition, even languages provide a automatic memory management, like Java, C# or LISP, it does not mean that the languages are immune to memory leaks. Although the memory manager can recover memory that has become unreachable and useless, it cannot free memory that is still reachable and potentially still useful. Therefore, modern memory manager semantically marks memory with varying levels of usefulness, which correspond to reachability, and frees an object that is rarely reachable.[5] However, the memory manager does not free an object that is still strongly reachable. Therefore, unless the developer cleans up references after use, no matter how robust and convenient the automatic memory manager is, all the programming errors that cause memory leaks are not eliminated.

Rational Purify® is one of automatic error detection tools for finding runtime errors and memory leaks in components of a program. Purify is a runtime analysis solution which is designed to help developers write more reliable code. The crucial functions of Purify are memory corruption detection and memory leak detection to ensure the reliability of programs and support runtime analysis[6].

1.2 Working Environment

Purify can automatically pinpoint runtime errors and memory leaks in Java™ and C/C++. It is available on Windows, Linux and UNIX. Purify also supports Visual C++ and all VS.NET managed languages (including C# and VB.NET). IBM provides Java™ and C/C++ versions of Purify with 15-day evaluation key on their web site [7].

1.3 How purify works?

Purify use patented Object Code Insertion(OCI) technology to instrument a program, inserting instructions into the program's object code. This enables to check the entire program, including third-party code and shared libraries even without the source code.

Purify keeps track of memory to determine whether it is allocated or not and initialized or not. Purify maintains a table to track the status of each byte of memory used by a program. The table contains two bits that describe 4 states of memory such as Red, Yellow, Green, and Blue and checks each memory operation against the state of memory block to determine whether the operation is valid. If the operation is not valid, an error will be reported [6][7].

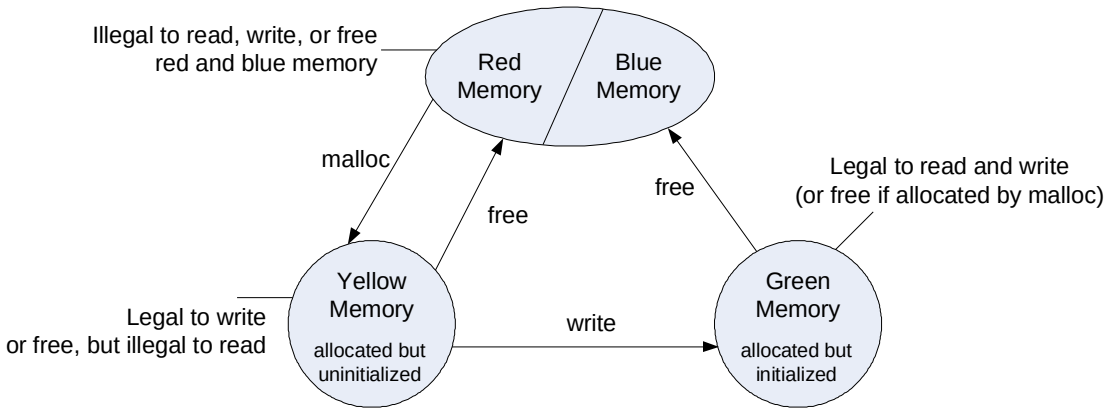


Figure 1 The state of memory in Purify

Purify sets initial heap and stack memory as Red state. Red state memory is unallocated and uninitialized. Either it has never been allocated, or it has been allocated and subsequently freed. It is illegal to read, write, or free red memory because it is not owned by the program. Memory returned by malloc or new is Yellow state. This memory has been allocated, so the program owns it, but it is uninitialized. It is illegal to read it because it is uninitialized. Green state means that memory is allocated and initialized. It is legal to read or write green memory, or free it if it was allocated by malloc or new. Blue state means that the memory is initialized, but is no longer valid for access. It is illegal to read, write, or free blue memory [6][7].

2 Studio project information

In the navigation system, POI data (POI DB) plays a role of input data, and is created by gathering Point Of Interest data such as building name, building information, telephone number, zip code, address, type of genre, location information (longitude and latitude), etc. However, it is not a map data. Those POI informations are gathered and integrated into one Microsoft Access database file (mdb) file in order to create the data file for car navigation system. Usually, one POI DB consists of more than 6 millions of individual POI data, so that the size of mdb file is more than 2 or 3 giga bytes.

Building POI DB file is conducted in 3rd party supplier, who is in charge of creating map data as well. While POI DB is established, cleansing process is absolutely necessary because some POIs can be overlapped or inconsistent by mistakes usually caused by human beings. When cleansing is completed, new version of POI is released and delivered to HMC multimedia team.

Once HMC receives new POI data, the engineer starts to convert POI DB file to binary file which can be applicable in the navigation system. This is because POI DB with mdb format itself can not be applicable in the embedded system with DVD media which shows bad performance in disk media. Thus, data format which is appropriate for the disk system is applied during the conversion process. The binary file (output data), therefore, has index data in tree shape (hierarchical form) and Meta data made up of POIs, and it helps navigation system to find POI in minimal disk access.

In the legacy converter system, data format and converter application is provided by external vendor, and they did not allow satisfactory modifiability (insufficient configuration input) in the converter, and the data format is not adequate for Korean alphabet system.

Therefore, HMC would like new application to equip with the configuration function and to resolve Korean alphabet issue. For example, the converter shall be able to configure the depth of index data, which may significantly vary the size of output data. The converter shall also be able to choose the search function to be shipped in output data. In addition, the converter shall allow the engineer to select/deselect Genres in output data. There are two type of Genre. One is top Genre; the other is sub Genre that belongs to top Genre. For instance, TGIF can be sub Genre, and family restaurant is top Genre. The following picture describes the production process of navigation data. The output binary data mentioned here is the final data produces in this process. (see disk picture in Figure 2) When it is complete, this data is burned in DVD, and distributed to end users with navigation system in the car.

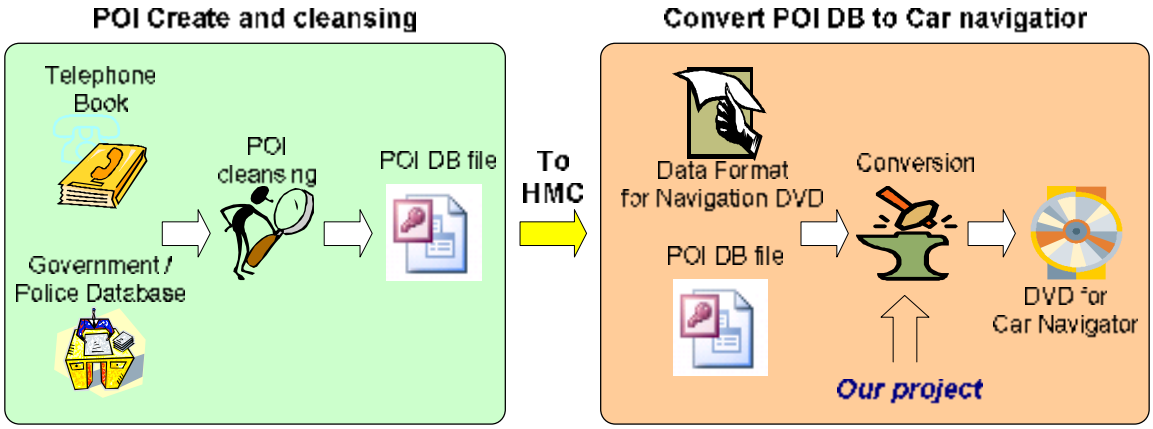


Figure 2 Navigation data production process

2.1 Applying Purify

Our studio project has technical issues to satisfy client's requirements. One of the most considerable quality attributes is "The system shall manage the gigabytes' input data, so that the hardware resources such as CPU and memory should support this process." As stated in the requirement, the application has to manipulate huge amount of POI data, and the size of POI data is about 1-gigabytes to 2- gigabytes. Therefore, careful memory management and precise estimation of memory usage is needed, because unexpected memory leak might occur due to incomplete garbage collection at runtime.

Moreover, the size of POI data will be growing, because more information will be added to POI data continuously. This will make the tree structure and computation grow up in the application. Therefore, the memory leak should be detected in advance and the logic which causes the memory leak should be changed. In addition, the performance of dealing with huge amount of data can be monitored by Purify. If there is a specific module which consumes memory extremely high, then it can be detected by Purify. Therefore, the module can be changed to other algorithms or other data structures. For example, a recursive tree algorithm can be changed to non-recursive algorithm with a linear data structure. Therefore, the application can guarantee not only the performance of system but also the reliability and availability.

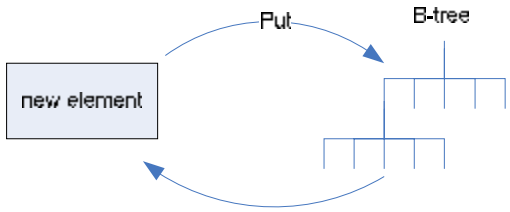
As mentioned in the chapter 1, Purify supports runtime analysis capabilities of memory corruption detection and memory leak detection. At the beginning, we considered Purify to detect memory leaks in two obtained source codes from website, modify them, and apply one of them to our studio project. In order to check the possibility of those strategies, two B-tree source codes written in different way are chosen and verified with Purify. However, it turned out that the memory leakage problem does not usually exist in the code unless the source code has significant mistakes or it has intentional memory leaks inside as the experiment is conducted in [11]. Thus, the strategy was changed to make use of Purify to measure the memory performance, analyze the code structure, and verify the performance. Once one of codes with better memory performance is chosen, the code analysis is performed with various functions provided in Purify such as function detail and call graph. After code analysis, code modification is carried out on the points where the modification possibly works. Finally, the verification for the modification is conducted.

3 Case Study with B-Tree Source Code

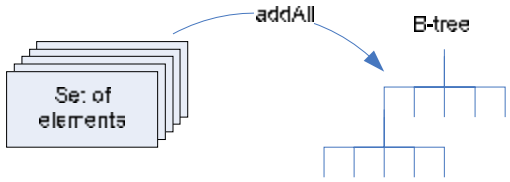
3.1 Source Code Introduction

Two source codes are obtained from the open source sites.[1][2] In order to compare the performance and memory consumption of codes, two candidate codes have following different aspects, assuming both provide same functionality in terms of balancing tree (B-tree) such as sort, insert, delete, split, merge, etc.

- Source A (obtained from [1]): Elements to be put into B-tree are inserted one by one. That is, the put function has to be called whenever new element is ready to be put into tree. Therefore, B-tree gradually increases the memory allocation of lattice as it more elements are put into tree.



- Source B (obtained from [2]): Elements to be put into B-tree are built into Vector form (Collection in Java), and referenced to B-tree module. That is, B-tree begins with allocating the memory with the size of elements, and adds those elements. Therefore, the entire memory allocation takes place for the first time.



In addition to insert method, there are differences in implementation such as how to insert, sort, and manage the tree structure, even though the functions are same. According to those differences, it is expected that both of source codes have different behaviors with respect to memory and performance.

3.2 Assumptions and Preparations

Since the purpose of this work is not only to tweak the memory and performance, but also to compare by means of Purify, both candidate source codes must have same functionalities such as sort and build B-tree. However, implementation way can be different, and the differences deviated from different implementation is one of applications of Purify. In addition, same amount of data (100,000 keys) are inserted in same way.

3.3 Select Source C

3.3.1 Source A

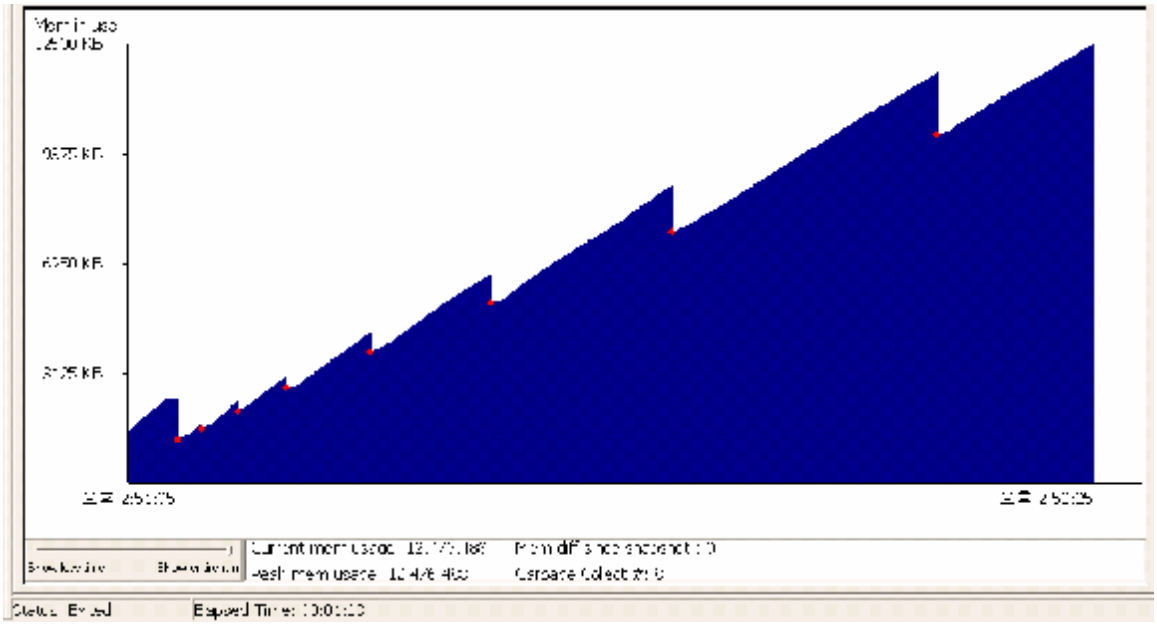


Figure 3 Source A memory status with 100,000 key data

Figure 3 represents the memory status of source A obtained in run time with 100,000 key data. As shown in the picture, the memory consumption gradually increases, and garbage collection (GC) occurs approximately 11 times during run time (see Red dots). The maximum memory required is about 12 mega bytes for this test.

Recalling the program structure described in the chapter 3.1, this picture illustrates that the application allocates more memory as the application accepts more key data. Thus, it is estimated that source A will expand the memory allocation in proportion to the number of key data. This behavior is very much same as illustrated in chapter 3.1.

This picture also shows the behavior of Java GC mechanism. This shows how Java virtual machine (JVM) uses system memory for its heap. The JVM may continue growing the heap rather than wait for a garbage collection cycle to complete and reduce the memory consumption.[3][4] However, this application does not look like having such a large unused memory, because there is no thread that consumes significantly large memory.

In addition, Purify shows the elapsed time, which must not be regarded as an execution time. This application shows 1 minute and 23 seconds, which is respectively higher than source B. However, the real execution time is nearly same.

3.3.2 Source B

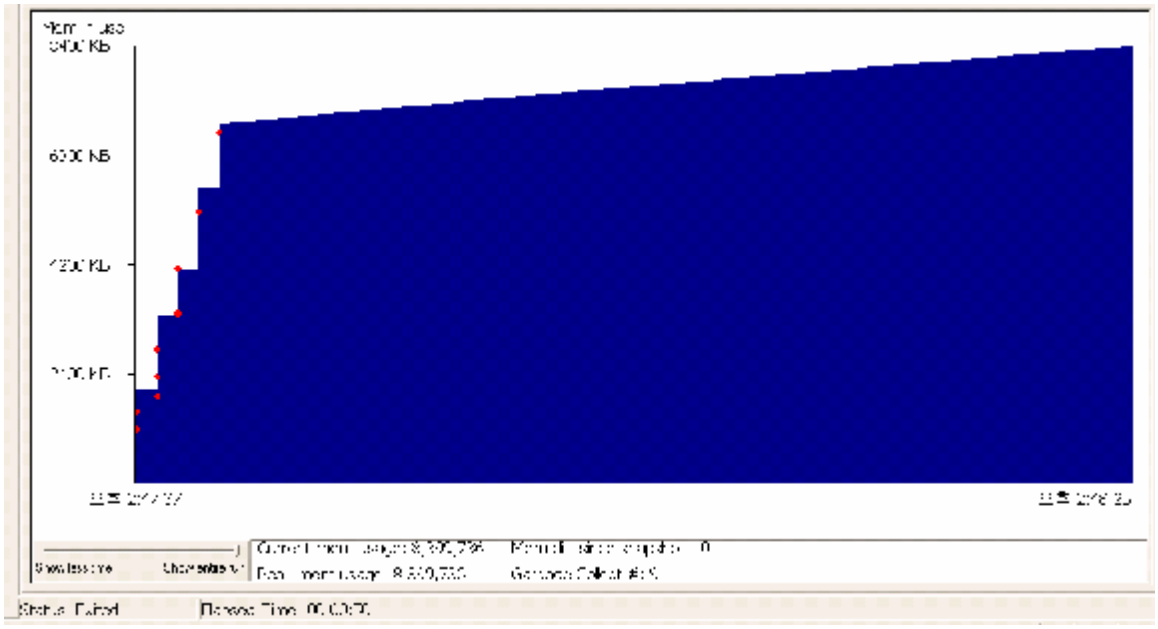


Figure 4 Source B memory status with 100,000 key data

Figure 4 shows memory consumption when source B is executed under same input condition as in chapter 3.3.1. Unlike source A, source B allocates all necessary memory at the beginning. The memory curve rapidly ramps up to approximately 8.4 mega bytes, which is remarkable smaller than source A. As mentioned in the previous chapter, the elapsed time shown in the bottom does not make sense, since it includes Purify overhead.

3.4 Select Source B and Further Application of Purify

One of the purposes of applying Purify in our studio project is to provide the guideline to select a source code which requires less memory as well as exploit the opportunities to tweak the performance of source code. Memory requirements are shown in the memory profile. In addition, call graph illustrates how the application operates in terms of call relationship among classes, and function detail shows how many calls take place and memory allocation in the class.

Thus, the following comparisons are done in order to choose a source code with better memory performance.

- Memory requirement: Source B shows better feature in terms of memory requirement. Comparing to source A, source B needs only two third amount of memory of source A.
- Call structure and recursive functions: Figure 5 is Source A call graph, and there are two recursive structures in insert function. Comparing to source B, which has only one recursive structure, source A looks more inefficient. Furthermore, there are more recursive structures (see red circles) in source A that are not revealed in this call graph, because those paths are not executed in this experiment.

Therefore, source B is chosen, and next is to tweak the performance further, because source B still leaves opportunities to be improved. For instance, by removing the recursive structure, it would be possible to improve the performance. However, it may cause another memory overhead by the local variable introduced while recursive structure is removed. Purify will help to compare the memory performance before and after changing the code structure.

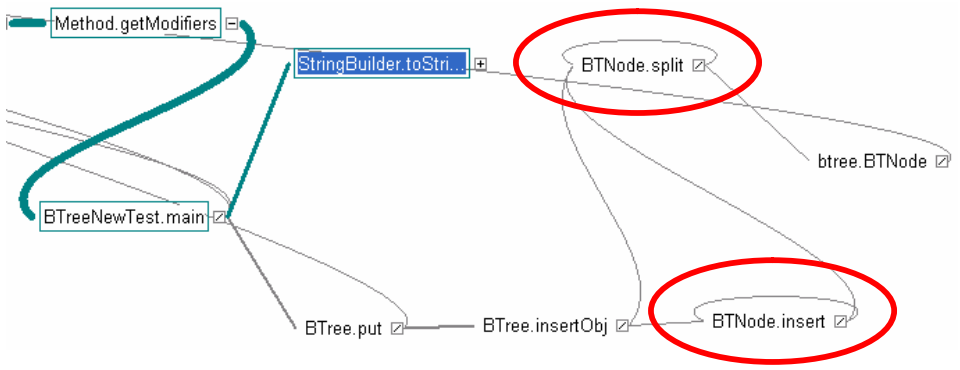


Figure 5 Source A Call Graph

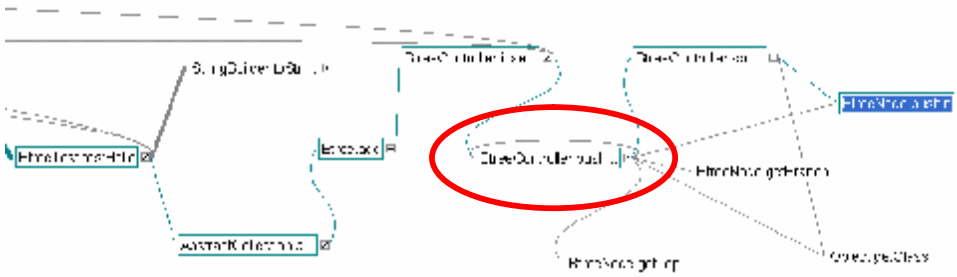


Figure 6 Source B Call Graph

3.5 Recursive Structure in Source B

It is said that recursive call is more intuitive and easier to code than non-recursive structure. However, it is not recommendable in memory and performance critical system, because it gradually takes up the stack memory and might cause memory overrun. Especially, when the recursive call takes place many times, it is more critical. On the contrary, by applying non-recursive call, it can be possible to reduce the amount of stack and number of function calls. Therefore, what we expect to see by introducing the non-recursive structure is to improve the memory as well as speed performance in source B.

In source B, recursive call is applied in tree traversing in order to find the empty node to put new elements. Even if the original source code preserves the top pointer where the previous element is put, and it guarantees to reduce the traversing effort significantly in next insert operation, it turns out that more than two times of recursive calls are necessary. (see Figure 7: 100,000 keys vs. 250,000 calls) The table on the left hand side is before starting recursive call. After recursive call begins (table on the right hand side), it is shown that additional 1.5 mega bytes memory are allocated and approximately 260,000 calls take place (see red lines). One strange fact is the entire memory consumption shown in the memory profile (8.4 mega bytes) does not include this additional 1.5 mega bytes memory. It seems like Purify does not take this memory into consideration. [11] explains similar experience that Purify does not detect uninitialized memory access in stack. Thus, it is necessary to look into the code carefully and compare with function detail in Purify.

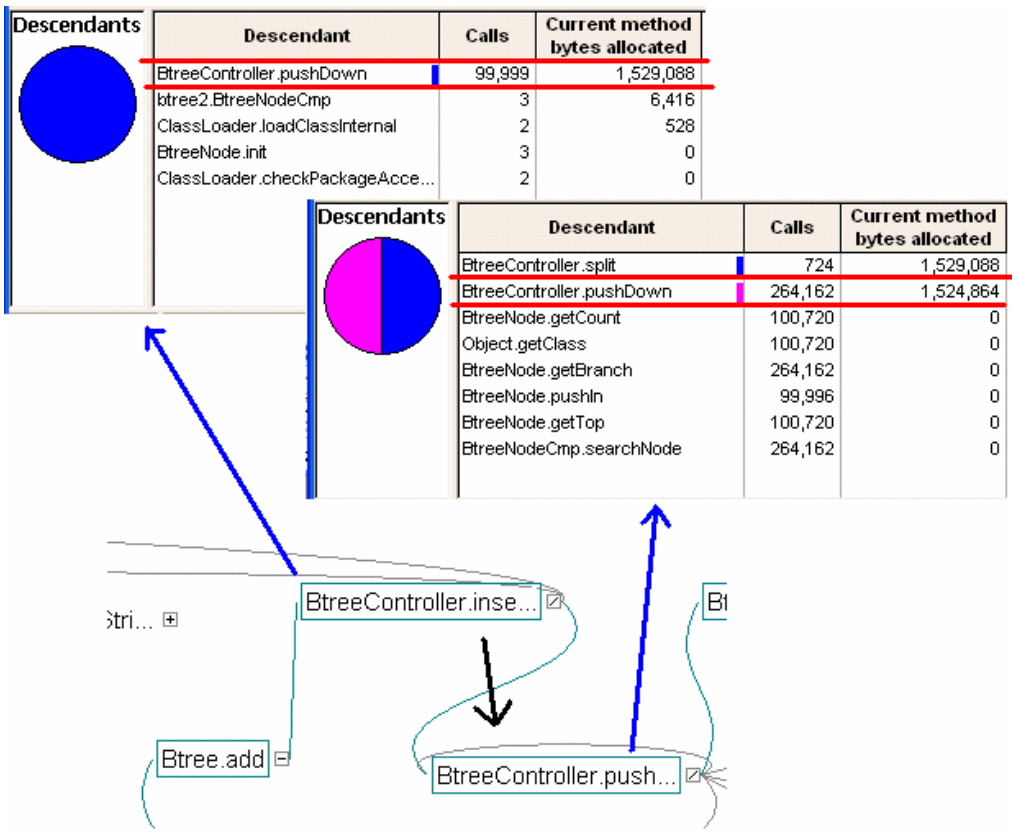


Figure 7 Number of Call and Memory Allocation in Recursive Call

3.6 Non-Recursive Structure in Source B

A local variable for TreeNode is employed instead of recursive structure. However, when replacing the recursive structure with non-recursive structure, a local variable or data structure is necessary, and it often causes another memory overhead. Therefore, the initial expectation to reduce the actual memory consumption is not achieved as shown in the Figure 8.

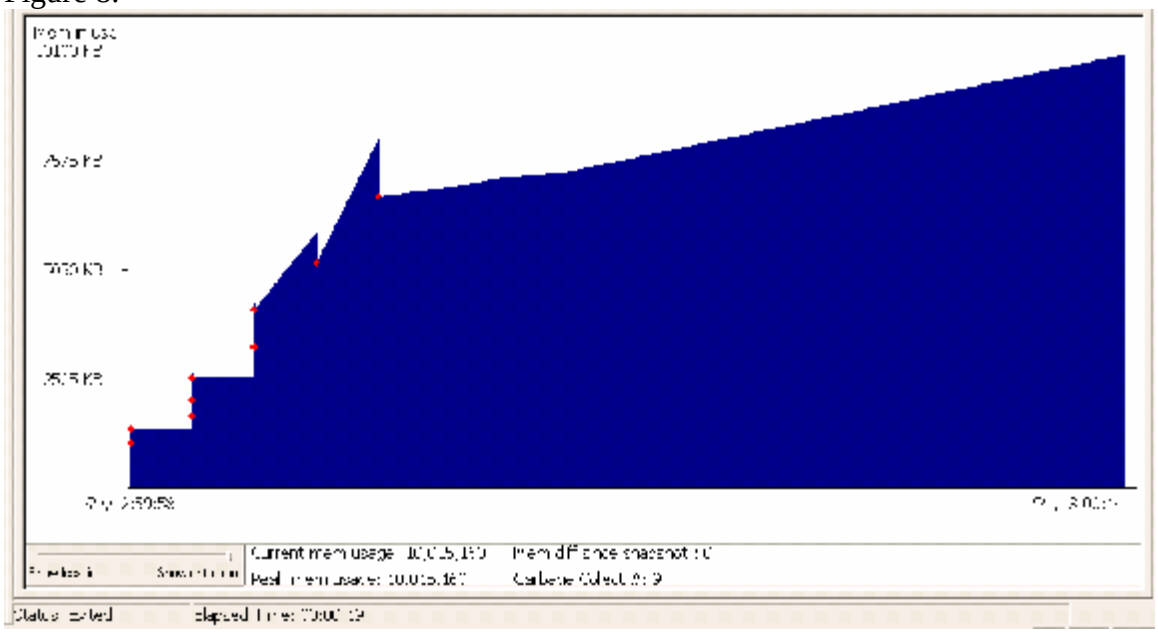


Figure 8 Source B memory status with 100,000 key data (Non-Recursive Structure)

The entire memory consumption is about 10 mega bytes, which is approximately 1.5~1.6 mega bytes more than the recursive structure is applied. This increment is caused by the local variable introduced instead of recursive structure. As for the execution time (not elapsed time), this modification requires only 281 ~ 313 milliseconds, which is faster than 400 ~ 430 milliseconds with the recursive structure. Even though, there is no improvement in memory performance, the execution speed is much better than the recursive structure.

In order to look into memory allocation and number of calls in detail, function details are shown in the Figure 9. Table in the left hand side is before the function where the modification is applied starts, and it shows a local variable named btree2.BtreeNodeCmp, which takes approximately 1.6 mega bytes. Instead, the modification does not show the additional function calls and memory allocation in stack memory. (comparing with before modification, see blue lines in the table in the right hand side)

As a result, it is possible to confirm the performance variation with Purify according to the code structure modification. For example, Purify shows the amount of local variable created in code restructuring and shows the recursive call is disadvantage to the speed performance. As for the memory increment by local variable, it is nearly same amount as the stack memory.

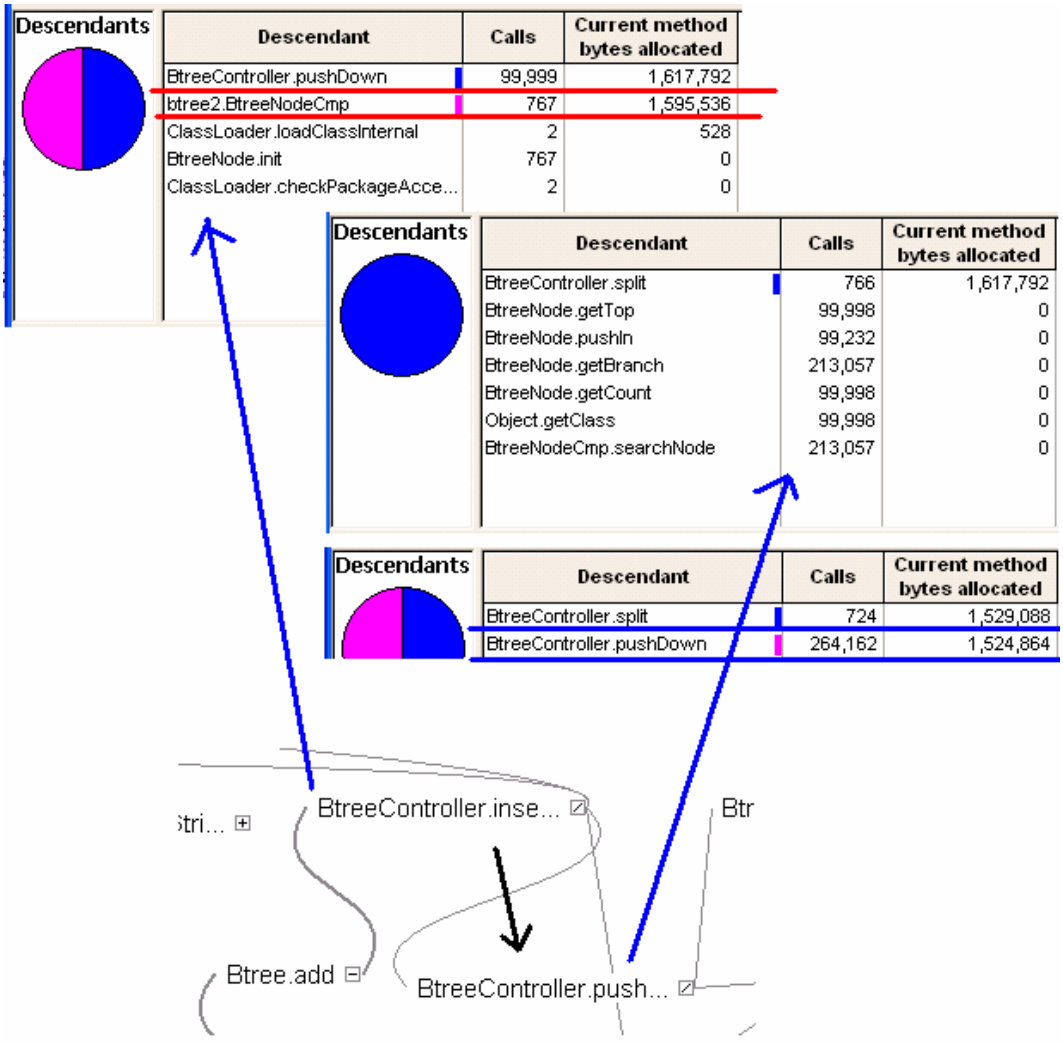


Figure 9 Number of Call and Memory Allocation in Non-Recursive Call

4 Application to Studio project

Purify seems useful in our studio project when this tool is applied to verify the performance of memory. Our case study shows the following possibilities to apply Purify in the studio project.

- Find the node or class where must memory is required: After run the application in Purify, Purify shows the call graph and function detail in addition to the entire memory profile. Call graph shows where most of memory is consumed by displaying the different thickness in the graph. In addition to the memory, call graph is helpful to identify the code structure. One of the interesting features is showing the number of call. This will be useful in improving the performance.
- Verifying the performance of different design patterns and implementations: Verifying and confirming the performance of software artifact is not easy, however, Purify shows the possibility for us to make use in verification of performance. For example, we are very interested in verifying the performance of sorting, tree traversing of tree algorithm in this project, because those will decide most performance of the application. However, it was not easy for us to find any efficient and convenient solution. Purify provides fairly easy and simple solution for this issue by measuring the memory and number of call.
- Coordinate the concurrent process: Since the conversion process might take several hours, it is also important to improve the execution speed performance as long as it does not jeopardize the memory performance. Thus, a concurrent conversion process is considered as one of the solutions to improve the speed performance, however, it would be critical with memory performance because of concurrent process might overrun the system memory. Hereby, Purify will be helpful to measure the maximum memory and peak memory usage of conversion process, so that it would be possible to coordinate the concurrent process to avoid the peak memory consumption.
- Determine the minimum resource to run the application: It is also important for the customer to know the minimum resource (memory) to run the application. Purify can easily measure the runtime memory requirement, and help to estimate the minimum resource.

5 Benefit and Drawback of Purify

There are benefits and drawbacks when using Purify. Benefits are illustrated in the chapter 4. Thus, only drawback will be discussed in this chapter.

- As mentioned earlier in the chapter 3.5, Purify does not show the memory allocated in stack while it is displaying the memory profile. It is not obvious why it is now shown, however, reference [11] mentions similar problem in stack memory. Therefore, a user may have to go through the call graph very carefully.
- Call graph does not show the path that is not executed. Since Purify is a run time analysis tool, it does not show the path that is not executed. Therefore, a user must know the important and critical execution paths, and force them to be executed by means of special condition or input configuration.
- Huge amount of system resource is required. When a large application needs to be run, it would be necessary to split the application, and run them partially. Purify needs huge amount of system resource, and it often notifies that the system memory is low.

6 References

[1] http://www.koders.com
[2] http://www.xs4all.nl/~jheyning/jeroen/Btree_java.html
[3] http://www-128.ibm.com/developerworks/java/library/j-leaks/ , Handling memory leaks in Java programs
[4] http://java.sun.com/docs/hotspot/gc1.4.2/, Tuning garbage collection with the 1.4.2 Java[tm] Virtual Machine.
[5] http://en.wikipedia.org/wiki/Memory_leak
[6] Rational Purify Installing and Getting Started, version 2003.06.00. S126-5312-00, publibfp.boulder.ibm.com/epubs/pdf/12653120.pdf
[7] http://www-306.ibm.com/software/awdtools/purifyplus/
[8] http://valgrind.org
[9] http://www.parasoft.com
[10] http://memwatch.sourceforge.net
[11] Zhao and Dong, Tool Evaluation of Rational Purify, 2002