Tool Evaluation of Rational Purify™

Gang, Zhao and Xin (Luna), Dong

1. Introduction

Memory access errors and memory leaks are some of the most difficult problems for programmers to solve. The bugs often only exhibit symptoms intermittently, making it very difficult to recreate and debug. No appropriate error messages are given for this kind of errors, thus programmers are just left confused by the bizarre results and may even not relate them to memory errors. In addition, the symptoms typically appear far from the cause of the errors. All of above make memory errors a big headache for programmers and make debugging them quite difficult and time-consuming.

Rational Purify is a run-time memory related error detection tool. It can discover almost all kinds of memory related errors and helps programmers to get to the root of the runtime problems. Its features and benefits include:

- It can automatically pinpoints hard-to-find illegal memory accesses and memory leaks in C/C++. Also, it can finds memory management issues in Java, C#, VB and .NET code. What's more, it can check errors in Web Server code including JSP and Jam Servlets.
- It can check not only users' source codes, but also libraries and even components, no matter whether there are source codes for them.
- It is available both for Windows and for UNIX. It is integrated with Microsoft Visual Studio 6.0 and Visual Studio .NET. It can quickly analyze executables, without any rebuilding.
- It permits programmers to control the error checking level for each code module.

In this report, we try to evaluate Purify in three aspects: functionality, performance and interface. In section 2 and 3, we focus on how effective Purify is in detecting memory related errors in C/C++ programs at runtime and in debugging garbage-collection related problems in Java. In section 4, we examine the overhead caused by Purify in execution time and memory consumption. In section 5, we briefly discuss its user interface. Finally, we talk about its role in software developing and draw the conclusion.

2. Purify for C++

C++ is well known for its high flexibility in memory control. On one hand, it brings convenience and strong power. On the other hand, it enhances the possibility for potential memory errors. Purify makes it much easier to find and fix these errors. It tries to find out all kinds of memory misuses and pinpoint the precise location. In this section, we firstly use a large amount of simple C++ programs to check what kind of errors Purify can detect and what it cannot. Then we test whether it also works well for MFC applications using a realistic MFC project.

2.1 How Purify finds memory-access errors

Before checking the capability of Purify to detect memory-access errors, we firstly have a look at the mechanism it uses in finding errors[5]. This can help us well understand or at least have a reasonable guess why Purify can or cannot do a certain kind of things.

Before execution, Purify copies the program and each library the program calls, and **instruments** the copies using Object Code Insertion (OCI) technology. The instrumentation process inserts instructions before each memory operation, including read, write, memory allocation and deallocation. The instrumented copies of each module are stored in the Purify cache directory. When rerunning a program, Purify saves time and resources by using the cached modules, re-instrumenting only the ones that have changed since the previous run. After the preparation, Purify starts the instrumented program and begins validating all the memory access.

During the execution, Purify maintains a table to track the status of each memory byte used by the program. For each byte, two bits are used to record whether it has been allocated and whether it has been initialized. The combination of the 2 bits identifies 4 states of memory, called red, yellow, green and blue in Purify. The state diagram and description table are shown respectively in Figure 1 and Table 1.

Purify checks each memory operation against the color state of the memory block to determine whether the operation is valid. If not, an error will be reported.

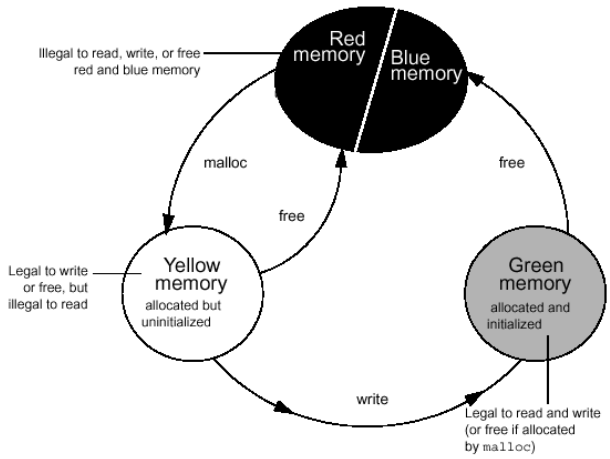

Figure 1 Memory states in Purify

Table 1 Memory states in Purify

State	Allo-cated	Initia-lized	Description	Illegal Operation
Red	N	N	1. Initial heap & stack memory 2. Guard zones around each allocated block and static data item 3. Freed uninitialized memory	Read, write and free
Yellow	Y	N	1. Stack frames on function entry 2. Memory returned by new and malloc	Read and Unmatched free
Green	Y	Y	1. Allocated and written memory 2. *data* and *bss* sections of memory	Unmatched free
Blue	N	Y	1. Freed initialized memory	Read, write and free

2.2 Purify for simple C/C++ programs

There are 5 categories of memory access errors in simple C/C++ programs. [1]

2.2.1 Array Bounds Checking Errors

Arrays can be allocated statically or dynamically. The former happens in the stack and the latter in the heap.

- **Dynamic Array Bounds Checking**

Both global and local dynamic array misuses can be successfully reported. Purify can detect the errors; report them as ABR/ABW messages; pinpoint precise error locations and allocation locations; and also give out how far the accessed memory is beyond or past the bounds. If the error happens in a function, both the place in the function and the invoking lines in the callers are pointed out. This is done by inserting red zones around allocated memory.

Since Purify keeps track of memory at the byte level, it can also detect errors if an int or long (4 bytes) is accessed from a location previously allocated as a short. These are also reported as array bounds read or write errors, as they are similar in essence.

Sometimes writing an array over bounds is not detected immediately and is reported later as an ABWL (late detect array bounds write). One example is that users input a longer string than the program has expected and allocated. This can not be detected until later use of the memory. Instead of reporting the exact place where the error occurs, Purify only gives out where the error is detected. Mechanism of stream operations is needed to be known to find out why Purify is confused here and cannot response timely.

- **Statistic Array Bounds Checking**

Static arrays are allocated in stack on function entrance. Like dynamic arrays, illegal static array accesses can lead to crucial mistakes. But Purify cannot detect them. The reason may lie in the guarding zone insertion around static arrays.

2.2.2 Memory Usage Errors

Besides array accessing errors, memory usage errors also include uninitialized memory read and copy errors, free memory read and write errors, and free mismatch errors.

- **Uninitialized Memory Use**

When the memory has only been allocated but not initialized, it is in yellow zone and cannot be read. Purify distinguishes between copies of uninitialized data, such as structure padding, and uses of uninitialized data in calculations. The former is reported as a UMC while the latter as a UMR. Again, the locations for both error and allocation are reported.

Here are three problems related to uninitialized memory use in Purify. The first is similar to static array bound checking. If a statically declared memory is not initialized, reading or copying it will not incur any errors reported by Purify. However, if the pointer for this memory is transmitted as a parameter of a function, any use of the pointer in the function will leads to an error. The problem is also related to how Purify color the memory in stack.

The other two problems are about the way Purify reports errors, thus just minor. One is that if an initialized memory is padded to an intermediate memory, and that memory is copied to another area of memory, UMC errors are reported twice instead of once. It is reasonable but redundant and may be confusing to users. It depends on how Purify handles with the copied memory. Specifically, whether it colors it green or the same color as the source memory.

Finally, sometimes Purify might attribute a UMR to the closing brace of a function. This is probably because one or more execution paths did not assign a return value for the function or because the value comes from an uninitialized location on the stack. As a result, users have to check all possible return locations.

- **Free Memory Use**

Free memory errors often happen when the program attempts to read, write or free a dangling pointer. The pointer points to a part of memory which has already been freed. So it is a blue zone error.

When such memories are read or written, FMR/FMW errors are reported. When they are freed, FFM errors are reported. Along with the error message, the precise places where the memory is allocated, freed, and attempted to access again are pointed out.

As freed memory can be allocated for other use and colored yellow or green again, Purify maintains a deferred free queue to record the ever freed memories so as to detect such kind of errors. Large queue length and threshold increases the chances of catching dangling pointer accesses long after the block has been freed and catching dangling pointer accesses to huge blocks of memory. This provides better error detection but at the same time takes up more memory at run time.

- **Free Mismatch Errors**

Free mismatch errors indicates that the program allocates memory from one family of APIs and then deallocates it from a mismatched family. For example, a memory allocated by `new` cannot be freed by `free`. Also, a memory allocated using one heap cannot be deallocated using another. These errors can occur both in yellow zone and in green zone.

When they happen, both the error message FMM and the codes of allocation and deallocation are reported by Purify.

However, there is another kind of mismatch. It happens when an array is allocated but delete is used to free it, or when a single memory is allocated but delete[] is used in deallocation. Unfortunately, purify doesn't take it into consideration.

2.2.3 Pointer Errors

Invalid or null pointers cannot be used in reading, writing or freeing. These operations can be so dangerous that sometimes the operation system itself breaks the program and gives out error messages.

- **Null Pointer Use**

When a pointer is assigned to NULL but read or written later, the system will stop the program and Purify will reports NPR/NPW with the error location. However, if the pointer is freed later, neither of them gives any responses.

- **Invalid Pointer Read/Write**

Invalid pointers include pointers pointing to memory on the stack, program codes and data sections, and also low 64k memory.

For system memory, none of the operations should be permitted. They cause IPR, IPW and FIM errors. Like the way to handle with NULL pointers, both the system and Purify protect from any violations of this area.

For memory on the stack and program data, it can be read and written but should not be deleted, which accounts for a FIM error. However, if several attempt of deleting occur in a line, sometimes Purify only reports the first one.

2.2.4 Other Stack Related Errors

Stack memories are allocated when entering a function and reallocated when leaving. Correspondingly, there are two kinds of memory errors related to stacks. One happens when not enough memories can be allocated on entering. Another happens when the reallocated memories are used again after leaving.

- **Stack Overflow**

Stack overflow usually happens in recursive functions if the termination condition cannot be satisfied and the program has no way to jump out. When the system is short of memory, it breaks current program and throws out an exception. Purify cannot detect it earlier but just report the exception again and again.

- **Stack Out of Bounds Read and Write Errors**

After leaving a function, the memories allocated for this function are colored blue and cannot be used any more. Or else it will produce a BSR or a BSW error. Purify reports the error locations but cannot report how far beyond the stack it is.

2.2.5 Memory Allocation Failure and Memory Leak

Another type of memory problem is memory shortage. The reason can be huge memory requirement or memory release failure.

- **Memory Allocation Failure**

If a huge block of memory is required at one time, the allocation cannot be satisfied and the system will throw out an exception. Purify can distinguish this exception from others, and gives out a MAF message as well as the location.

Another possibility is that the program keeps asking for small blocks of memory, and at last makes the system run short of memory. In this case, Purify makes things worse. Before the program eats up the memory, Purify runs out of virtual memory and has to be paused manually. What's more terrible, if the program is run under debugging mode in Visual Studio, things turn out that in the end the project cannot be closed without using Task Manager.

- **Memory Leak**

Memory leak can leads to shortage of swap space and finally slowing down and crashes. Besides its jeopardy, it is infamous for the difficulty to be detected. One of Purify's biggest contributions is that it can locate memory leaks precisely. During process shutdown, Purify scans heaps for leaked memory indicated by yellow zones and green zones before calls to HeapDestroy. As memories in stacks are released automatically, they will not lead to memory leak and are not needed to be examined. If some memories are allocated but not freed, the MLK information will be presented with the allocation places. Furthermore, if for some blocks, Purify cannot find any pointers to its start, but there appear to be pointers pointing somewhere within the block, then a MPK message will be used to indicate a potential memory leak

Again, some problems with memory leak exists in Purify. One is that if there is no return at the end of the main function, some of the leaks will not be detected. Another is, if exit(), ExitProcess(), or TerminateProcess() is called, and these variables contain pointers to blocks of allocated memory, the memory is considered still in use and is not reported as a leak. However, if instead the program return from main() and all local variables go out of scope, additional memory leaks might be reported.

2.3 Purify for MFC programs

Nowadays a lot of software written in C++ is essentially extensions for the Microsoft Foundation Class(MFC) library. The main differences between this branch of C++ programs and simple C/C++ programs include that the projects are larger, the invocation relationship is more complex, and more Windows API and handles are used. To test whether Purify works well for these programs, we use a middle-sized project that interactively draws B-Deboor curves as a test case.

Before testing, the project had already been compiled and could draw B-Deboor curves as expected. After running Purify, one array bounds error and five memory leaks were detected. Besides, a bad parameter error was reported. The Purify reporting window is shown in Figure2. With the help of Purify, it's not difficult to find the array bounds error and two of the memory leaks. While fixing the memory leak errors, some new problems were introduced intentionally or accidentally. Every time Purify gave us a helpful and timely response. Finally these mistakes were fixed, while the bad parameter error and three other memory leaks were left. At the end of the test, one handle mistake and one reserved memory misuse were led into the program. Purify detected the mistakes and reported them.

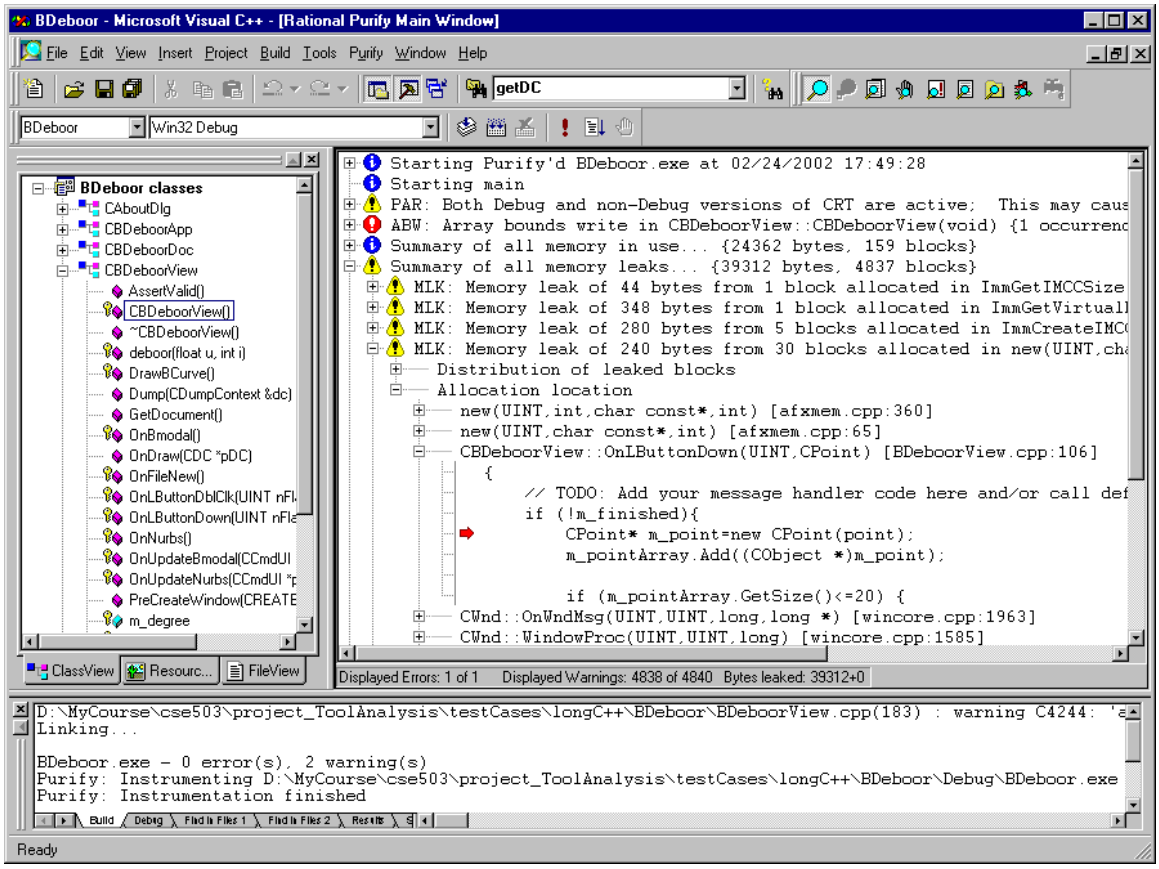

Figure 2 Purify Report for BDeboor Project

The whole process shows some basic characteristics of Purify in detecting and reporting memory related errors in MFC programs.

- Firstly, the error location reports for MFC programs include much more functions to record the invocation trace. Usually the inner functions are API core functions that handle with memories; the outer functions are also API core functions that handle with events; and only a couple of the functions in the middle are defined in our own programs. Purify can detect API functions even when the source codes are not available.
- In some cases, Purify cannot point out errors directly and accurately due to the complexity of the program. Although it still gives a lot of clues for locating those mistakes, programmer's good knowledge of the class structure is crucial.
- Some memory errors are reported for almost all MFC programs, even the simplest ones produced automatically by MFC wizard. One of them is the bad parameter use error, which says both Debug and non-Debug versions of CRT are active. The others are 3 memory leaks, including 44 bytes allocated in ImmGetIMCCSize, 348 bytes in ImmGetVirtualKey, and 280 bytes allocated in ImmCreateIMCC.
- When some reserved pointers, such as the document pointer got by GetDocument(), are deleted, under debug mode the program will be broken by the system and Purify will report a null pointer read error. However, the location given for this error is the first use after the pointer is deleted and usually far away

from the deletion. Users may be confused by it and it can be difficult to find the error. However, if running it under release mode, the program can run well and no errors are reported by Purify.

- Purify also reports handles in use after execution. For some handles that are not released on exit, they can lead to some resources leaks. As a simple test, we included a call to GetDC() without a matching call to ReleaseDC(). Purify generates a list of unreleased handles on exit, including this one and also a number of others that are not resource leaks. Thus it's the programmer's job to distinguish them and remove the leaks.

3. Purify for Java

Besides detecting explicit memory problems in C/C++ programs, Purify can also find out potential memory leaks in Java. By sacrificing some flexibility, Java eliminates many memory errors in C++. However, memory problems still exist in Java. In this section we first describe these problems, and then use an example to illustrate how Purify can help in detecting these problems. Finally, we show a by-product in which Purify is used as a tool to study memory allocation strategy in Java.

3.1 Memory problem in Java

Thanks to Java's great design, the common illegal memory accesses in C++ cannot disturb Java programs at all. Firstly, Java prohibits the use of pointers. This protects the system from unintentional damage caused by pointers, and also semantically eliminates all errors related to pointers. Secondly, Java Virtual Machine (JVM) performs much more strict checks in run time. It can detect almost all kinds of illegal memory accesses and throw out run time exceptions. Furthermore, JVM uses garbage collection to collect unused memory periodically and automatically. In this way, the traditional memory leaks are completely prevented. Table 2 lists the memory errors and presents how they are resolved naturally in Java.

Table 2 Memory Errors and Resolutions in Java

Error Types	Solved in Java by
BSR, BSW FMR, FMW IPR, IPW	In Java there are only references but no pointers. So these problems cannot happen.
UMC, UMR	Java does not process memory directly. Any object will be initialized using its constructor at creation. So these problems cannot happen in Java.
ABW, ABR	IndexOutOfBoundsException: Thrown to indicate that an index of some sort (such as an array, a string, or a vector) is out of range.
NPR, NPW	NullPointerException: Thrown when an application attempts to use null in a case where an object is required.
MLK	The garbage collector. The memory blocks with no references will be cleaned by the garbage collector in the future.
MIU	This is the real memory leak in java application.

However, there are still inefficient memory uses in Java. They can greatly damage performance and even cause program to crash. A common problem is that the memory consumed by an application increases stably over time. This can be a result from [3]:

- Adding objects to collections or arrays but forgetting them after that.
- Resetting the reference to another object. If the routine in which the reference is reset is not called, the object stays in memory and will not be garbage collected.
- Changing the state of an object when there is still a reference to the old state.
- Having a reference that is pinned by a long running thread. Even the object reference is set to NULL, it cannot be garbage collected until the thread terminates.
- Using system resources that are not freed automatically. For example, Abstract Windowing Toolkit (AWT) for Sun Java will not be cleaned by the Garbage Collector and needs to be freed manually by using the method dispose().

Thus, memory leak in Java is defined differently from that in C++. In Java, memory leak is the memory garbage occupied by the objects that would not be referred any more according to program's logic, but fail to get rid of all references to them. To remove this kind of leaks, we must distinguish unintentional memory waste from intentional memory use. This is very difficult in large application. Fortunately, Purify can make it easier.

3.2 Using Purify for Java "memory leak"

Purify also uses the technique of instrumentation to profile the memory usage of a given Java application. With the profiling data, Purify can tell which methods and objects monopolize large chunks of memory that the garbage collector cannot free.

To evaluate the effectiveness of Purify™, we designed a sample with intentional memory leaks inside. In a thread class, we introduced a bug that causes stable increased memory use. Normally, the bug is out of the execution path, thus the thread object works well. However, it can be triggered by a signal. Our application produces 10 threads of this kind, and only one of them is chosen randomly as the victim of the bug. As a result, which particular object is chosen is not known in advance

To use Purify in profiling Java memory usage, the basic steps are:

- Run Java application with Purify.
- Take a snapshot when memory usage is stable.
- Execute codes that may cause memory leak and then take another snapshot.
- Compare the two snapshots and identify the methods that may give rise to memory problems.
- Pinpoint the objects and the references that prevent the objects from being garbage collected.

Figure 3 is the screen copy when Purify is doing profiling. The application shows that the abnormal thread is thread 5. Figure 4 shows that by differentiating the two snapshots, Purify also succeeds in finding out where the problem is. The thickest line denotes the object responsible for the memory leak. In function detail view (Figure 5) we

can find that thread 5 allocates 90K memory while others only 2k, which confirms our suspicion. Additionally, Purify can locate the source code and help programmers fix the problem. (Figure 6)

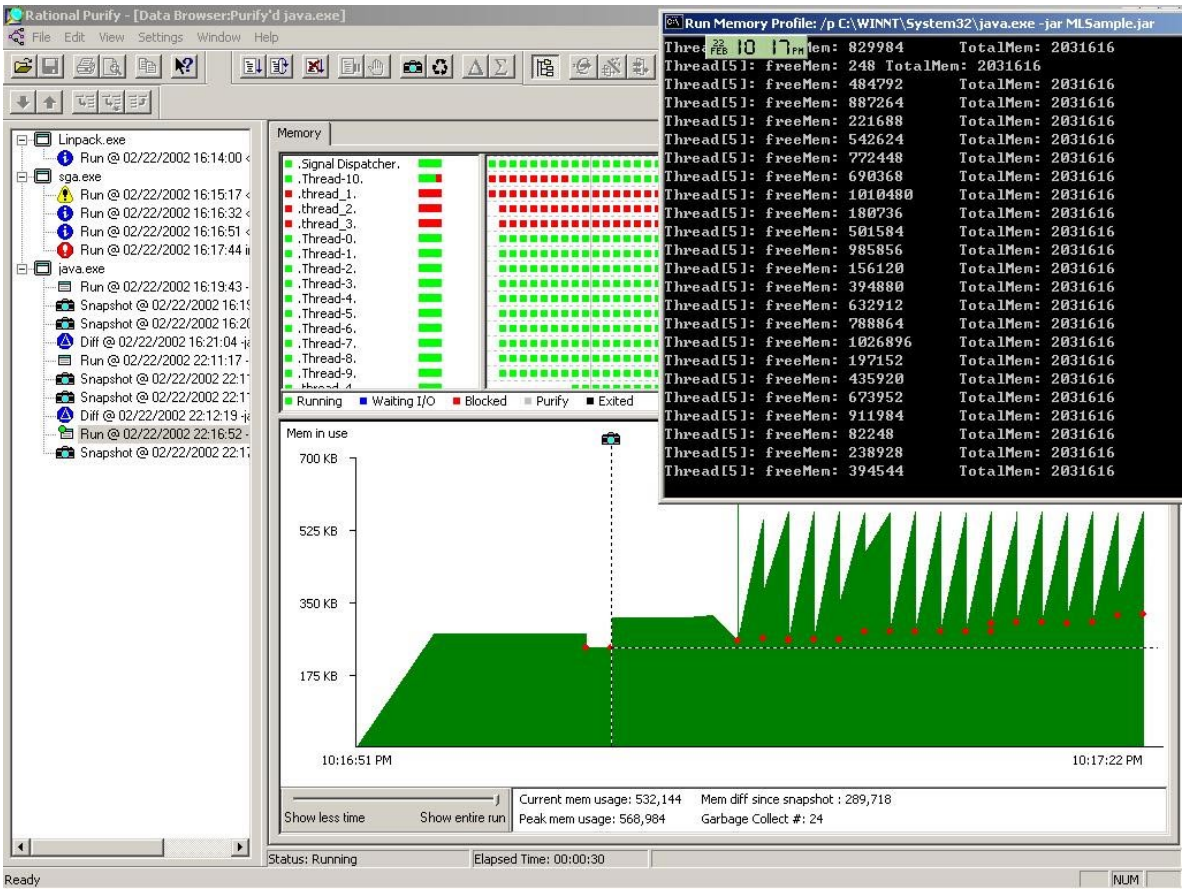

Figure 3 Memory profiling with Purify

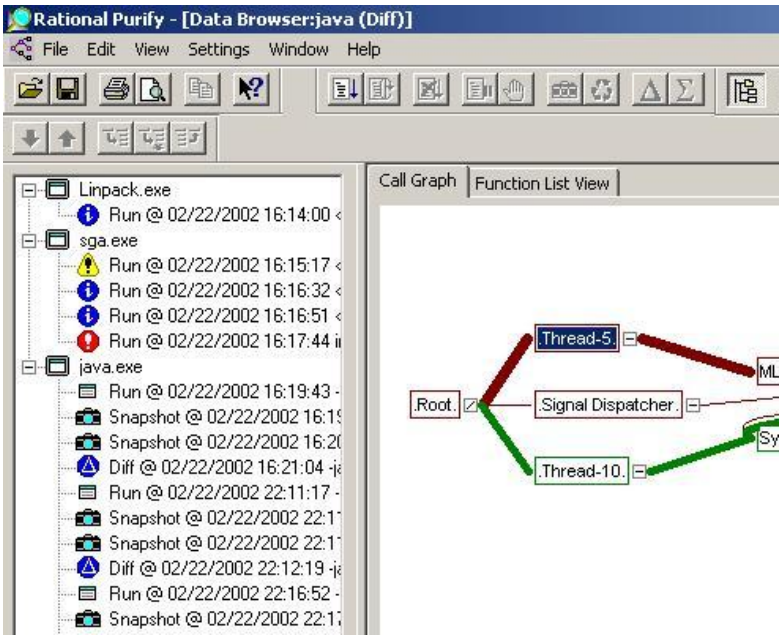

Figure 4 The thickest line indicates potential memory leak

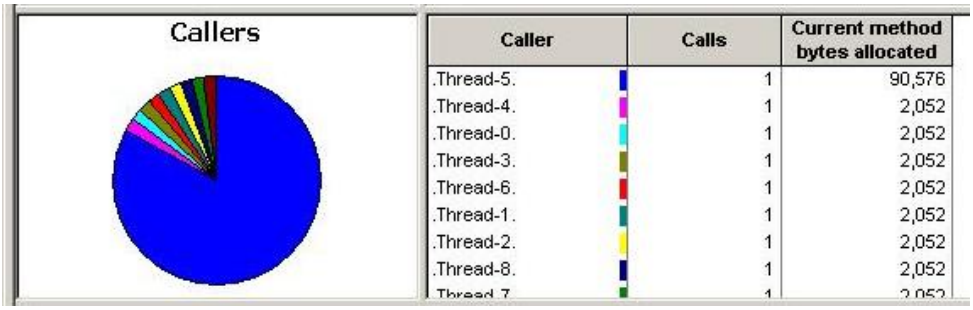

Figure 6 Memory usage of each thread object

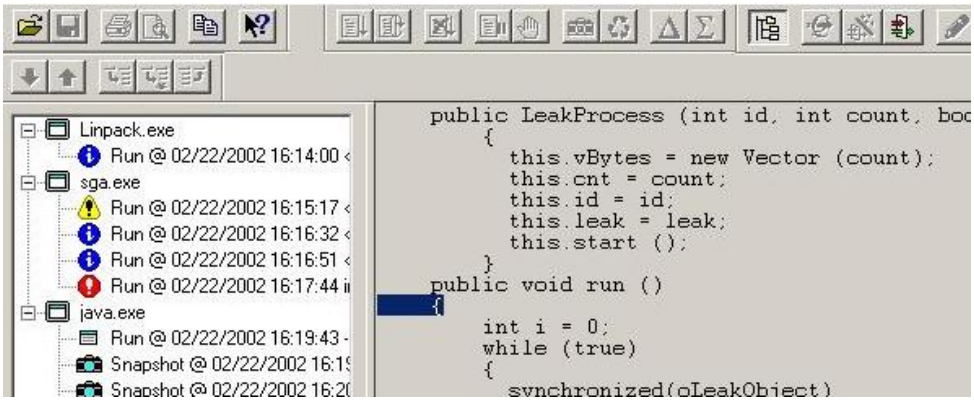

Figure 5 Come back to source code

3.3 Using purify to study memory allocation strategy in Java

A by-product of Purify evaluation is that the memory diagram profiled by Purify reveals the adaptive memory allocation strategy in Java.

When starting an application, JVM allocates a certain size of memory as visible memory to the application, which is larger than the current requirement to a certain percentage. When more memory is needed, JVM first tries to allocate memory from current visible block. If the required amount is larger than the current visible block, JVM supplements more memory. The increment is not constant but also keeps increasing.

We are interested in the amount of each increment that JVM allocates to satisfy the memory requirement. In our experiment, we varied the size of allocation requirements from 512 Bytes to 1 Million Bytes. Figure 3-7 shows the total memory size of the visible block each time. Figure 3-8 shows the increment size. The figures show an interesting result: even though the memory requirements vary greatly, JVM increases the visible total memory in the same pattern. Beginning at about 1MB, the increment doubles each time.

With Purify's memory profiling function, we can get the data of memory usage directly from the memory view. Even though using runtime API could also reveal the memory usage, but Purify bring us a visual impression to the background problem with its diagram of memory profiling.

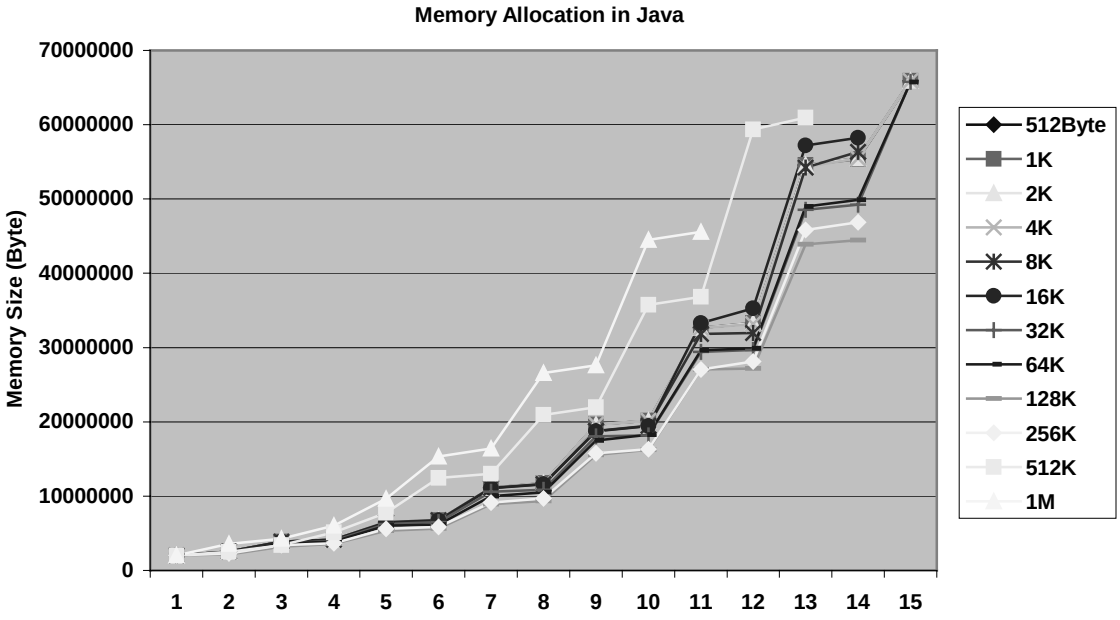

Figure 8 Total memory allocated to sample application

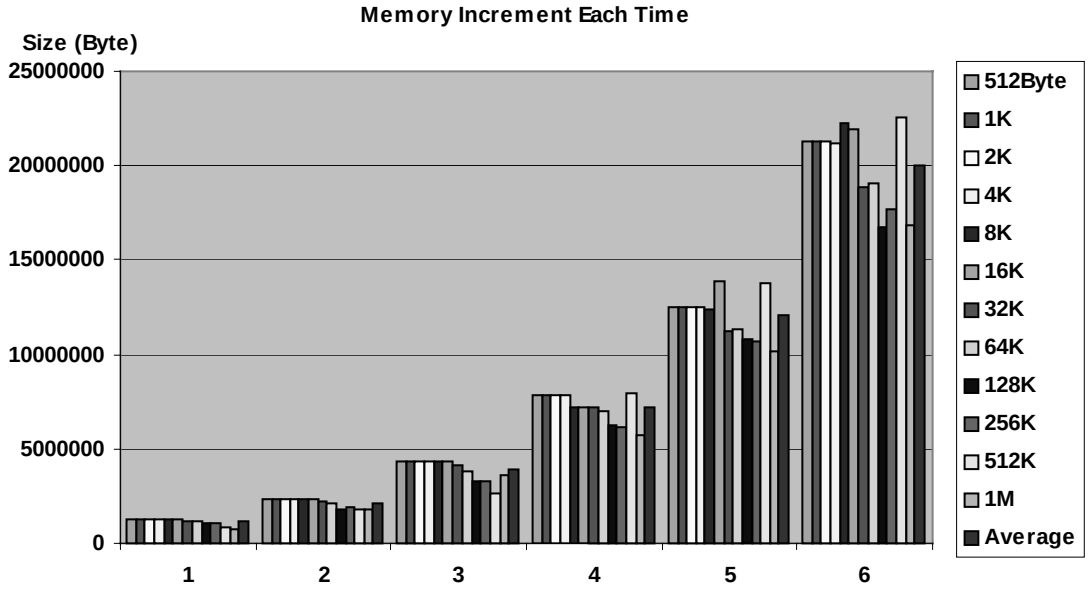

Figure 7 Memory increment each time

4. The Performance of Purify

When looking for errors in programs, performance is a secondary issue to features, however, it is still an issue. The performance hit when running with Purify is significant. [4] As Purify runs its own private instrumented versions of the application and maintains a memory state table, both the running time is prolonged and the system memory is grabbed as well.

In this section, we observe the overhead by examining execution time and memory consumption. Our test cases vary in run length, error types and error numbers. Originated program performance is compared with that under Purify monitoring.

Firstly, we choose a C++ program that does matrix multiplication. There are two versions of the program: one uses static array; the other uses dynamic array, in which there are much more memory uses. Both of the versions are run in different matrix sizes of 10, 50, and 300. Table 3 shows the time spent on executing these programs with Purify and without Purify. Figure 9 shows the comparison. From them we can see, although they are just simple programs, a large amount of overhead is introduces. More overhead is introduced to programs using dynamic memory than those using static memory. In addition, it's easy to understand that as the matrix size goes up, the overhead decreases.

Table 3 The overhead of Purify in C++ program

	Execution time without Purify	Execution time with Purify	Ratio
30static	188	1766	9.39
100static	2891	21610	7.47
250static	16891	75125	4.45
30dyn	47	1864	39.7
100dyn	1688	21489	12.73
250dyn	20546	143374	6.98

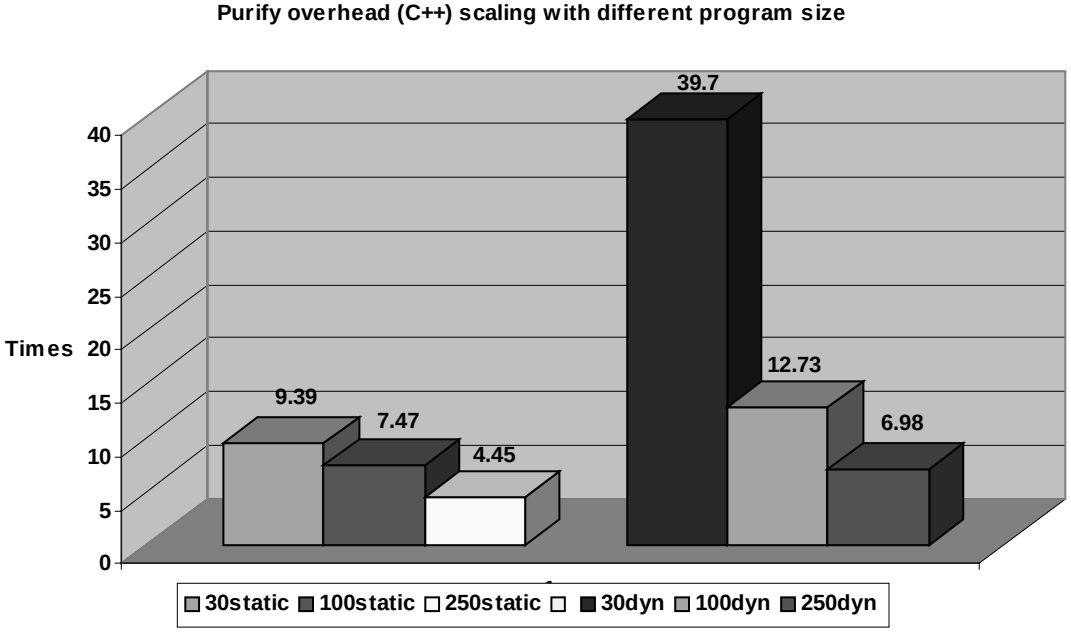

Figure 9 The overhead of Purify in C++ program

Secondly, we make ten copies of the mid-sized matrix multiplication program using dynamic arrays. In each of the first four copies, we lead into one standard error types: read out of bounds error, read uninitialized memory error, read freed memory error and memory leak. In another four copies, the same errors are brought in but occurred much

more times. In the rest two copies, each has two kinds of errors. Table 4 gives out the execution time and Figure 10 shows the time ratios of the original correct program and the wrong programs. The ratio varies in a wide range from error type to error type. Usually, more memory errors need more detection time. However, although the running time under Purify is often more for error programs than correct programs, that without Purify's detection is not. So no error doesn't mean lower overhead.

Table 4 The overhead of Purify with different amount of errors

	Execution time without Purify	Execution time with Purify	Ratio
Original	1688	21489	12.73
ABR less	2047	22032	10.76
ABR more	2891	23125	8
MFRless	63	22015	349.44
MFR more	63	22585	358.49
NUL less	2906	22656	7.8
NUL more	4328	35890	8.29
LEAK less	2859	22078	7.72
LEAK more	1672	21828	13.05
ABR+LEAK	2907	23156	7.96
NUL+LEAK	4312	37093	8.60

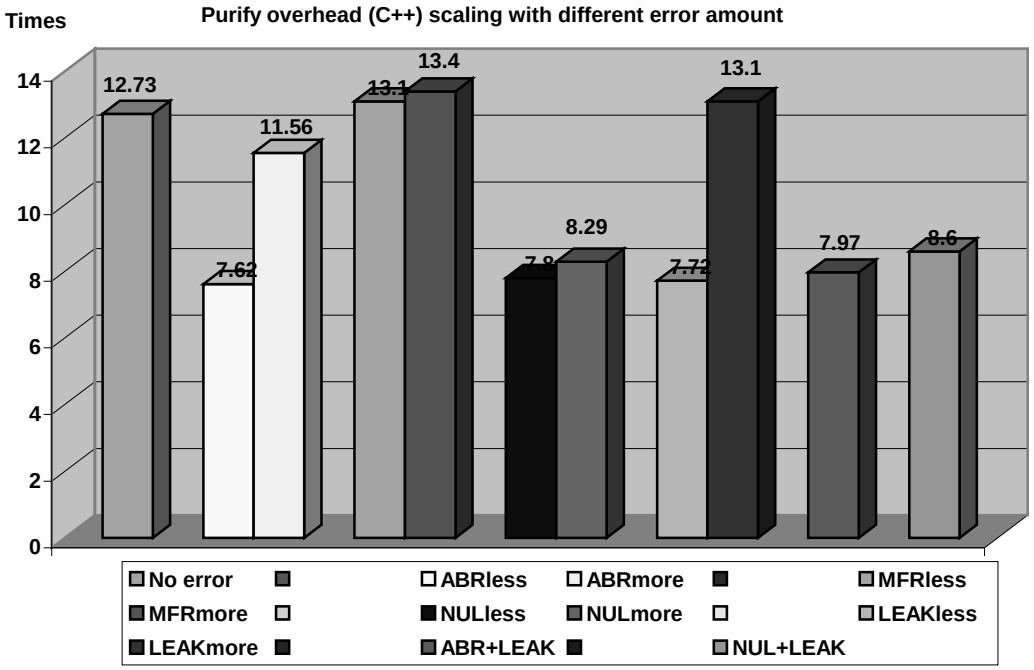

Figure 10 Purify overhead (C++) scaling with different error amount

In addition, we use a small benchmark from CS department of University of Wisconsin to quantify the performance degradation of application under the profiling of Purify for Java. From the benchmark we selected 5 programs of different code size and

run time. The performance is measured in two aspects: execution time and memory consumption.

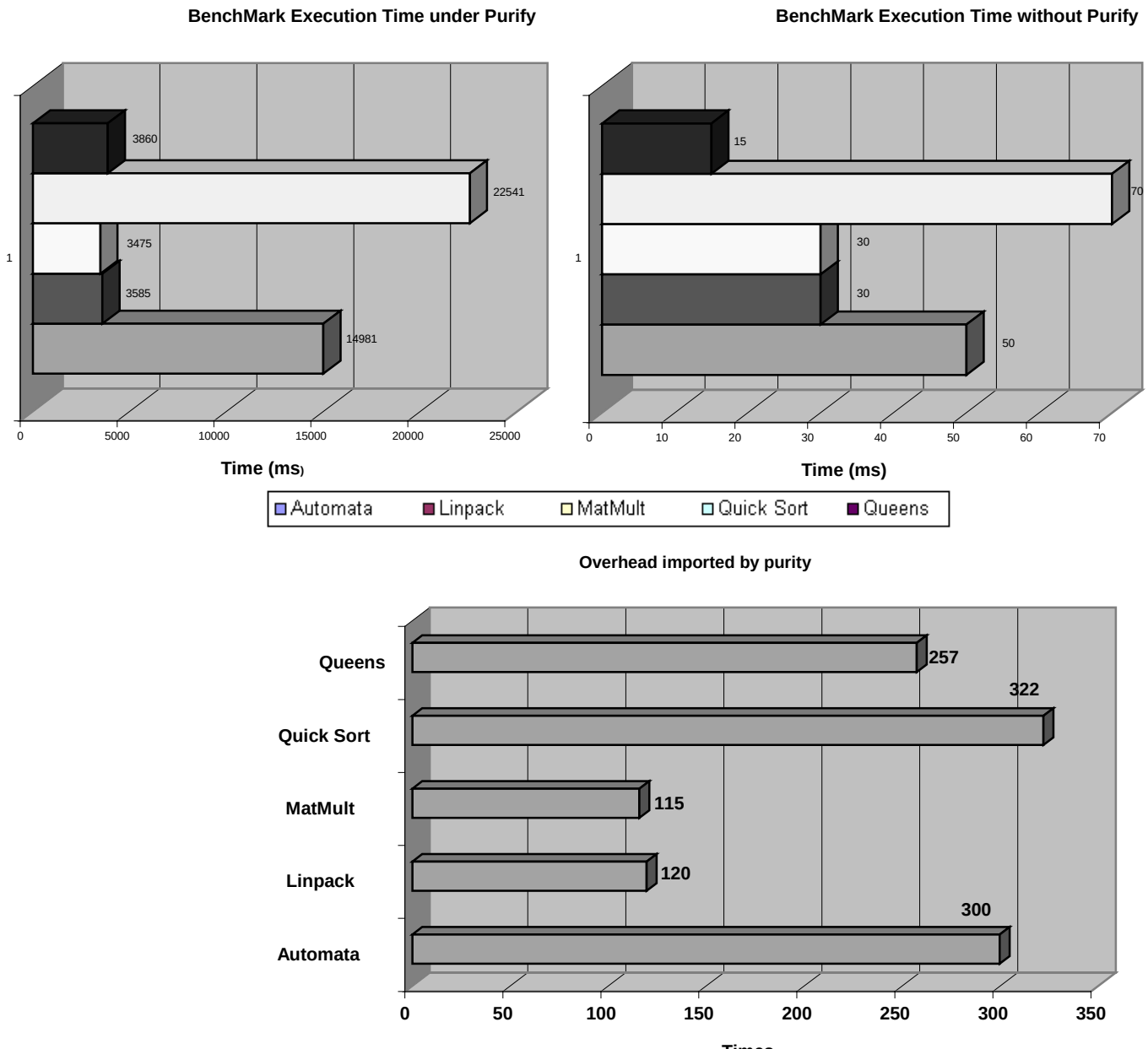

Figure 11 Overhead in term of execution time

As shown in Figure 11, Queens, Quick Sort and Automata are some kind of memory intensive, which contains frequent memory accesses and a lot of objects at the same time. For these applications, the overhead brought by Purify is about 300 times of the execution time. MatMult and Linpack could be considered as computation intensive, in which the memory accesses are less frequent and the number of objects is moderate. The overhead introduced by Purify is about 100 times. Again, this result is reasonable considering that purify monitors memory accesses.

From Figure 12 we can see that in average the profiling consumes 150% more memory than original execution. This limits the largest size of the applications that Purify can profile. Actually, when profiling large applications, Purify often gets crashed, although it is just the large programs that are more likely to contain potential memory problems.

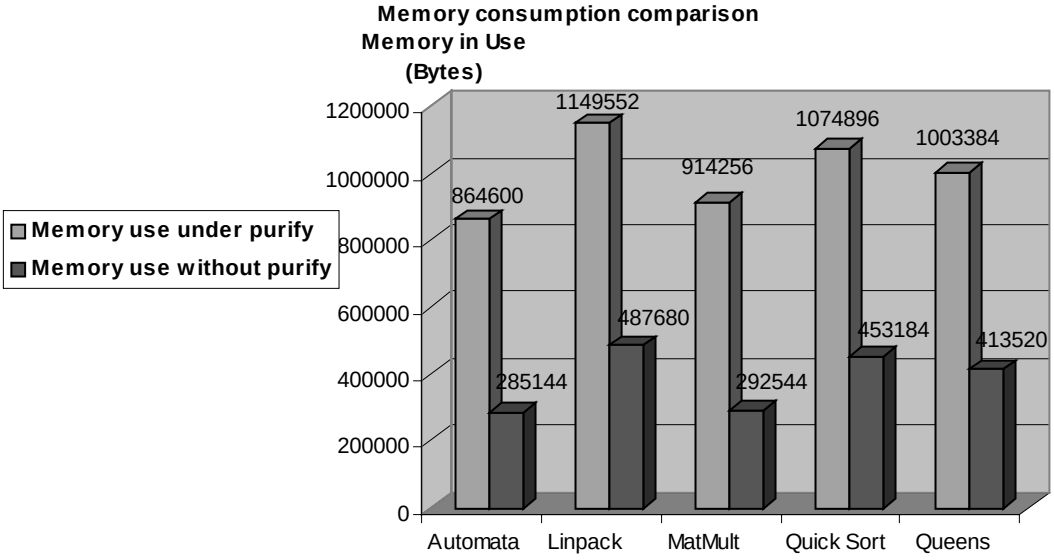

Figure 12 Memory consumption in each case

5. The User Interface of Purify

Purify is integrated into Microsoft Visual Studio 6.0 and Visual Studio .NET. Besides, it can be used as a standing alone Windows application or in command lines. The high consistence with Windows program style makes it easy to learn and easy to use.

Purify presents memory errors and usages in a more visual way. For C/C++ programs, it organizes the information in a tree structure. Different symbols and colors are used for different information types. What's more, it can bring the programmers back to the source code where the error occurs when clicking the corresponding error information. For Java programs, it generates memory profiling information, which contains 4 views: memory, call graph, function list view, and object list view. The graphical charts are expressing and clear.

Besides, Purify is easy to control. Programmers can adjust the level of checking on a module-by-module basis. They can design filters in details and also choose whether to stop the execution when errors are detected.

Finally, the command line of Purify enables it to be invoked in a test script. So we can integrate Purify with those test tools to launch a bunch of profiling or detecting commands one time in batch mode. Purify has a complete set of optional parameters in

command line version to enable all functions of it could be accessed through command line.

6. Conclusion

Rational Purify helps to locate and eliminate those memory-related errors which can be fatal or corrupting. It is helpful to enhance both the functionality and the stability of the software. Rather than waiting until the final release, it should be used early and often throughout the development process.

Although Purify is powerful and convenient, it cannot substitute for the developer's expertise. In debugging using Purify, at least three things are needed to be done by the programmers themselves.

1. To design different execution streams to cover all of the routines in the program. Purify validates memory accesses in run time. In other words, this is done dynamically and errors can still be left in the codes that are not executed.
2. To tell real memory leaks and resource leaks from those that are needed to be in use on exit. Purify is still not smart enough and often mix them together.
3. To track down the error locations with the help of Purify. Sometimes Purify cannot point out error locations accurately. A good knowledge of the class structure can be very helpful in doing this.

In the evaluation, we are impressed by the strong power of Purify. However, we have to admit Purify is not perfect. While it can play an important role in software development, it still has large room to be improved.

1. The biggest and most annoying problem about Purify is its stability. When instrumenting C++ files, it makes the whole system crash from time to time. This can happen either on large projects or on small projects, either after it has run for a long time or immediately after it is started, either when the project is compiled for the first time or when it has already been compiled for several time. We also tried to avoid the crash by deleting all the files in the cache folder whenever running Purify, it seems to be helpful but crash still happens some time.
2. Another problem is its scalability. As it eats up huge amount of memories, it cannot successfully detect large applications. When we tested Java programs, it crashed from time to time when the application was large. Fortunately, this time it was itself but not the whole system that went down.
3. The use of cache folder is also a problem for Purify. Although it brings higher performance and saves time and resources, it occupies more and more places in hard disks. When the project is ended, the instrumented files are still left in the hard disk and so need to be deleted manually.
4. While Purify succeeds in detecting dynamically allocated memory misuses in C/C++, it cannot handle with static memory well. Array bound accesses and uninitialized memory access in stack can neither be detected.

5. Finally, in Java memory profiles, when the memory leaks are small, it's not detectable from the report views. Although these leaks are minor and is not a big threat to the system, it can become a problem after long-time run.

Reference

1.Rational Software Development Company, Rational Purify for Windows "http://www.rational.com/products/purify_nt/index.jsp".

2.National software Testing Laboratories, Performance Testing of Rational Software's software product Purify ,"http://www.rational.com/media/whitepapers/pnt-nstl.pdf"

3.Goran Begic, Memory profiling in Java, Rational Software white papers.

4.Goran Begic, Run-Time Debugging with Microsoft Visual Studio and Rational Purify

5.Rational software white papers