Analysis and Detection of Performance Bugs

Buting Ma Shuoren Fu Yuantong Yu
University of Michigan

Abstract

Performance bugs do not cause program errors, but would impact speed or responsiveness. They could degrade performance, increase power consumption, and even crash the entire system. Moreover, performance bugs are much more difficult to detect than functional bugs, because they cannot be decided by examining the output.

In this paper, we studied 48 performance bugs and their root causes. For each root cause, we observed distinguishable behavior features that could imply potential performance bugs. Based on our observation, we developed methods to detect performance bugs, and build prototypes to verify our methods. Based on our findings, we learned that 1) Recreation of threads can cause less threads to be runnable. 2) Static analysis can filter out initialized but not used classes. 3) Static analysis can filter out unused attributes in container elements. 4) Statistical analysis of statement execution count can be used to infer infinite loop.

1. Introduction

Debugging performance issue is hard. Unlike functionally bugs, performance bugs are very hard to locate, since the program runs correctly. Performance bugs are harmful that it sometimes decreases the efficiency by orders of magnitude. It also wastes system resources, and affect the rest of the system.

Existing performance bug detection techniques mainly rely on static analysis. While these methods are effective at detecting bugs associated with loops [2, 3], they cannot capture the runtime behavior of the programs.

We collected performance bugs from bug databases of several popular java open source projects, including log4J, tomcat, PDFBOX, Lucene, and Groovy. We searched performance bugs that cause "CPU spikes", "CPU 100%" "slow" "high memory usage". We focused on the solved ones, so we can learn the root causes from developers' discussion and patches.

We sampled 48 performance bugs and classified they into 5 categories (with 13 sub-categories), according to their root causes. The result is shown in Figure 1. We detail each of them in section 2.

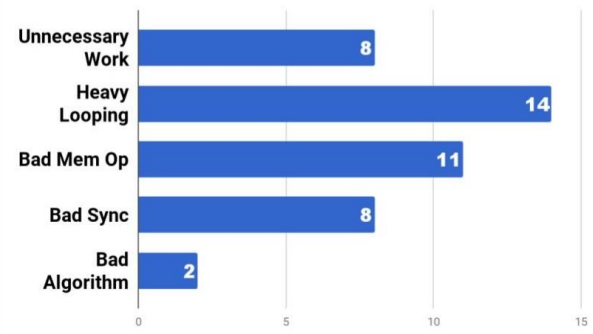

Figure 1. Distribution of different performance bugs. Others including dynamic compiling features, unmatching API, etc.

We choose to work on Java projects because it is relatively easy to profile. JVM could also provide us more opportunities of detecting performance bugs and even fixing them dynamically. Previous work on java performance bugs have shown this [6]. Although we find the performance bugs from Java programs, the root causes are universal and fundamental. The root causes involve abstract system designs and can happen in any programming languages.

2. Root Cause Analysis and Classification

2.1 Unnecessary Work

This category introduces performance bugs that include unnecessary work. By eliminating them, we can reduce the execution time of a piece of program.

2.1.1 Recreation

In this type of performance bug, programmer repeatedly create and delete the same data structure, typically within a loop. The solution(optimization) would be resetting and reusing this data structure. This bug can be implicit. For example, declaring a container within loop, and passing function parameters by value, can both cause recreation problem. Some deferring mechanism, such as copy-on-write, can help eliminate this problem, and sometimes compiler could help resolving it in optimization steps. However, the recreation problem still causes a lot of performance bugs. In this problem, if deleting always happens before recreating, memory consumption would not increase. However, the processes of deleting and recreating add too much overhead.

One specific situation is the recreation of threads. This usually happen in server applications. For each request, the program create one thread to handle it. This works fine under light load, however, under heavy load, it will be a disaster. Too many thread will devour all the memory and introducing massive context switching overhead. The solving is to use a thread pool, and we will address this in section 3.1.

Detection Suggestion: monitor the events of creating and deleting certain data structure, including the execution of constructor and destructor, memory allocation and release. If we see these events happen too many times or too frequently, we can suspect that recreation problem exists.

2.1.2 Redundant Calculation

This problem is similar to the previous one. Due to bad design, the program will repeatedly do calculation on the same input and get the same output. One example would be redundant traversal bug [2]. The optimization is to do the calculation once, then store and share the results (trade space for time).

This problem could happen on distributed computing system that the input is distributed, and the calculation is performed on each node, rather than do calculation on one node and distribute the results. However, this could be done intentionally, either to reduce communication (input is smaller than output), or to eliminate "straggler" problem (MapReduce).

Detection suggestion: in profiling, monitor the input and output of a certain function across all executions. If the input and output are the same across too many executions, the redundant calculation bug could potentially exist.

2.1.3 Unnecessary Initialization

Unnecessary can happen in two cases.

In first case, unnecessary initialization happens when a class is initialized but never used later. Initialization is done by special functions such as the default constructor, the self-defined constructor, and the self-defined initialization function, etc.

In second case, initialization is done to guarantee the correctness of operations on the data structure. For example, assigning a large array to all zero before using it. However, we have other mechanisms to guarantee this, therefore this initialization can be considered unnecessary.

The first case can be relatively easier to detect. We will show an approach with static analysis in section 3.2. However, for the second case, it is not easy to detect automatically. Because to decide the unnecessity (redundant guarantee), the semantics should be understood, and this is a human's job.

2.1.4 Unnecessary Check and Update

The performance bug happens when the program so frequently pulling remote resources. However, the remote resource does not change often, therefore a lot work on checking the remote resources is wasted. This usually happen in the initialization phase of application.

The optimization is to reduce the frequency of remote checking. Dynamic rate control mechanism can be very helpful.

2.2 Heavy Looping

Heavy looping is the most common performance bugs. Many things can induce heavy looping.

2.2.1 Busy Waiting

Busy waiting is a bad programming practice. A thread loop on itself waiting for other threads, instead of yielding its resources to others. This thread remains in runnable state, doing nothing but wasting power, and introducing more (and useless) context switch.

The behavior of busy waiting is similar to infinite loop, we develop a method to detect both of them, which is detailed in section 3.3.

2.2.2 Retry Lost Connection

This performance bug happens when the program repeatedly retries opening a lost connection. This will be a serious problem within a bad network condition, because even if the retry is successful, the reopened connection is usually unstable and very possible to be lost again, causing more retrying.

One way to resolve this problem, similar to 2.1.4, is to adjust the retrying rate. Similar to congestion control in TCP, we could limit the number of retry and reduce the "aggressiveness" of retry during the process.

Detection suggestion: for this type of performance bug, we could monitor the network traffic. If abnormally many failed handshaking is observed, we can suspect the existence of this performance bug.

2.2.3 Infinite Loop

Infinite loop can also be considered as a functional bug. Usually it is due to lack of error handling. Unexpected input will trap the program in states that exit condition can never be triggered.

Theoretically, we can confirm an infinite loop if we observe repeated program states. Symbolic execution

will be helpful in generating the test cases that expose the infinite loop problem. However, this approach fails when environment interactions are involved.

Therefore, we developed dynamic approaches using profiling. We will detail them in section 3.3.

2.3 Bad Practice on Memory

Memory is also a cause for some kind of performance bugs. This kind of problem is closely related to unnecessary initialization but manifested in memory outage. From the bugs that we found, we can see that server-side applications such as Apache Tomcat, who needs lots of caching mechanism to ensure performance, or such as Apache Ant, who involves a lot with file input and output in order to log the build information, are very likely to suffer from memory related performance bugs.

2.3.1 Too Much Data in Memory

One large category of the problem is caused by caching too much unnecessary data in memory. In the bugs that we found, programmers tend to cache data information in array or map like data structure or output too much data in the file stream without flushing it to the disk. In this case JVM will throw an *OutOfMemoryError* exception, which cause the program to cease to function, or JVM will start to trigger garbage collection too often to create more usable memory space, which will cause too much background work and user application start to incur performance degrade. We will discuss how to detect such "put too much things in memory" problem statically in section 3.4.

2.3.2 Cannot be Garbage Collected

Another interesting category of memory related performance bugs is memory leak. Some long running modules may have a static reference to objects from other modules which should be garbage collected as the modules is undeployed. However, they cannot be garbage collected since the reference count is not zero. The possible solutions are to re-initialize the classes that contain such static references using *Class.forName* method. Future works can be done to provide a general solution to this type of problem to reload certain classes when necessary.

2.4 Bad Synchronization

Performance bugs in multi-thread programs can be caused by bad synchronization mechanism. Too many or too few synchronization, or deadlock can introduce performance bugs.

2.4.1 Too Many Synchronization

Synchronization is necessary to guarantee correctness of multi-thread program. However, too many synchronization causes degrade the performance. Those synchronization mechanisms will break parallelism and force the program to run in serial. Examples includes misusing unique lock instead of read-write lock, and using coarse-grained locks make critical section too large.

Detection suggestion: monitoring the activity of threads. If the number of runnable threads is significantly smaller than designed value, this problem possibly exists.

2.4.2 Too Few Synchronization

Too few synchronizations can also cause performance problems. Multiple threads or processes could compete for resources, causing problems like cache contention or false sharing. One example is thundering herd. It occurs when a large number of processes waiting for an event are awoken when that event occurs, but only one process is able to proceed at a time. Therefore, most processes will wake up, do nothing, go sleep over and over again, adding massive system overhead.

Detection suggesting: also monitoring thread activities. If there are more runnable threads than desired, or the system overhead is too large, then we may need more synchronizations.

2.4.3 Dead Lock

This is a very common performance bug. Similar to infinite loop, it is also a functional loop. The fundamental reason of deadlock is that the lock dependencies form a loop, and no one on the loop can make progress.

There are a lot existing researches on how to detect, avoid, and resolve deadlock problem. The simplest idea is to see whether there are long blocked threads, and on which locks do they block. If these locks form a loop, it is a deadlock.

2.5 Bad Algorithm

This problem is subtle. In practice, whether an algorithm is good or bad depends on the input size. Some algorithm may have lower complexity. However, it runs slower on small-size input. Sometimes programmer prefer algorithms with higher complexity because those algorithms are typically more direct in idea and easier to implement.

Machines can help little in this problem, since machines don't understand algorithms and their working conditions. Detection this problem relies on code review by programmer.

3. Detection Method and Evaluation

Based on our findings in previous section, we chose four types of performance bugs to detect: 1) Recreation instead of reuse, 2) Unnecessary initialization, 3) Heavy loop, 4) Bad practice on memory. The source code of our prototypes can be found at https://github.com/sean-shuoren/Detection. We evaluate our detection methods on synthesized workloads, and leave the detection on real-application workloads in the future work.

3.1 Recreation

Among the many reasons that cause unnecessary recreation, we studied the detection of recreating threads. The detection of recreating data structures, streams, and containers can be detected similarly. It will be discussed in section 5.

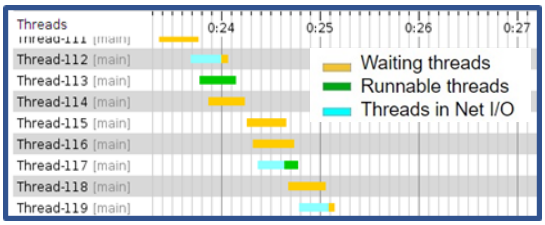

Figure 2. Thread Recreation

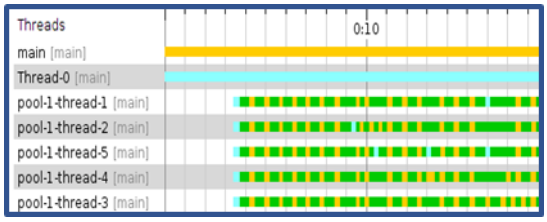

Figure 3. Thread Pool

To detect recreation of threads, we use JProfiler to inspect the thread activities. Figure 2 and Figure 3 depicts the runtime information of our synthesized program. The program is run on Ubuntu 12.04, on top of an Intel i3-4005U CPU, with 1.7GHz frequency and 4G RAM. Recreation of threads causes more threads to be blocked and less threads to be runnable. Table 1 shows that using the thread pool with carefully tuned size, we can achieve comparable latency and throughput as that of thread recreation.

	Multithreaded	Thread Pool of Size 5	Thread Pool of Size 10
Average Latency (ms)	3747	4525	3431
Throughput (requests/sec)	12.33	10.45	12.83

Table 1. Results of average latency of and throughput

3.2 Unnecessary Initialization

Figure 4 shows an example of unnecessary initialization. Class MammalInt has a data member of class DocFooter. It also has an init() as initialization function, and eat(), travel() as non-initialization functions. Similarly, class DocFooter has init() as initialization function, and paint() as non-initialization function. In the MainEntry class, a non-init function (function eat()) of class MammalInt is used, but the member DF inside MammalInt is only initialized. We therefore mark the initialization of class DocFooter at line 15 as unnecessary.

```
1   Class DocFooter{
2     public void init(){}
3     public void paint(){}
4   }
5
6   Class MammalInt {
7     DocFooter DF;
8     public void init(){}
9     public void eat(){}
10    public void travel(){}
11  }
12
13  Class MainEntry{
14      MammalInt m = new MammalInt();
15      m.DF = new DocFooter();
16      m.init();
17      m.eat();
18  }
```

Figure 4. An unnecessary initialization bug that initializes an instance of class DocFooter but never used it later.

To detect the bugs with unnecessary initialization, we statically parse the java bytecode of the original program (Figure 5). The additional input file specifies the names of initialization functions of classes. If a class is only invoked by initialization functions, these initializations are considered as unnecessary.

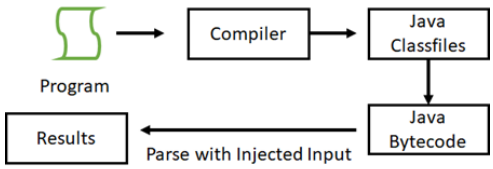

Figure 5. The process of statically detecting unnecessary initialization bugs.

3.3 Heavy Loop

There are two approach we can detect possible heavy looping. The first is on timing, and the second is on counting. Each will give us clues on infinite loop, and we can have better results using both of them together.

The first approach is learned from website fastthread.io [7]. The idea is to take several thread dumps with certain interval, e.g. 5 seconds. Stack traces are analyzed. The same stack trace across consecutive dumps implies that the correspond thread is trapped in a loop.

This approach can be improved. The stack trace includes source code line number, and we can learn which lines belong to the same loop by source code analysis. Therefore, the test condition can be extended to whether a runnable thread remain in the same loop across consecutive stack traces.

The second approach is developed from instrumentation. The idea is simple: counting the execution of each statement. If the execution count of certain statement is abnormally high, or larger than that of nearby statements in orders of magnitude, then the heavy looping bug could possibly exist.

Figure 6 is an example of how we can use this approach to detect busy waiting. We use a line coverage tool called Cobertura. We can see that the execution count of the busy waiting loop is in billions, extremely larger than other statements, which only have 2 execution counts.

```
73                  public int worker(int x)
74                  {
75
76  1262580213              while(taskQueue.isEmpty());
77
78                2         Task task = parseTaskString(taskQueue.poll());
79                2         task.execute();
```

Figure 6. Execution Count

Both approach could only imply the buggy behavior, and cannot guarantee the existence of heavy loop bug. These two approaches could generate warnings on where is the possible bug, which possibly include a lot of false-positives. It is programmer's job to decide whether it is a heavy loop bug and how to fix it.

3.4 Bad Practice on Memory

For this section, we talk about the detection of the first type of memory related performance bugs, which is putting unnecessary data in a container. We choose to detect and mitigate this type of problem in compile time since the usage of such container is largely dependent on the program behavior and there is less flexibility to control the whole program. We extract the program logic in the buggy Tomcat module and write a simplified test case to develop and test our detection mechanism.

We compiled the test cases programs to class files and input to a modified Soot compilation framework. As Figure 7 shows Soot first parse the program and transfer it into Jimple intermediate representation. After that it builds up the control flow graph and applies transferring, optimization and annotation is global scope. Then it applies such process in local scope, which is in every class method. And finally output the optimized class file. We add a new transform layer in wjtp, which is the whole-jimple transformation pack. We choose to do it in a global layer since we have to consider the container usage in all methods.

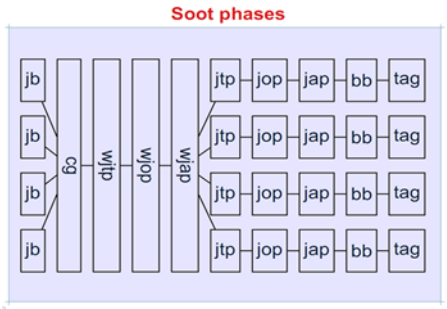

Figure 7. Soot phases.

In the new transfer layer, we first check all the fields of each classes, add such field to a suspect set if the field is a container type in java.util package, which includes list, queue, set and map. For now, we are only checking all the private containers so that all of their usage is expected to be find in the code that was input to Soot. Then for each container that we have, we check all the methods in its declaring class and find all the usage of the elements of that container. Since in a typical Java program, get and set method is used to access the member variables of a class, only the methods of these elements are recorded. And a set of element fields that is accessed in each called method as we go through them. We take a union of these sets and if there are significantly less fields than all fields of that element, then we conclude that such container contains unnecessary data and can be optimized.

After we have detected those containers, we can optimize it by modifying the program's Jimple intermediate representation. We can create new classes for such containers. All the necessary fields declaration and methods are copied from the old element class declaration. When an old element is put into the container, an instance of a newly defined class is replaced. The data of necessary fields is copied from the old class using get and set methods.

We are now able to identify such containers that stores unnecessary data from our test program. However, we do find limitations when we apply this detection mechanism to real world large applications and implementing the mitigation mechanism. We will talk about these limitations in section 5.

4. Related Work

Classification. Jin et al. provide a comprehensive study of characteristics of performance bugs [1]. They mainly summarize three general root causes. A rule-based performance-bug detection is performed to match performance anti-patterns. Unlike their work, we present more detailed classification of performance bugs, and use syntactic analysis to detect bugs.

Static and Dynamic Detection. Several projects use program analysis to detect performance bugs. CARAMEL project uses static analysis to detect performance bugs related to wasteful loop calculation [3]. Olivo et al. uses static analysis to detect performance bugs related to collection traversals [2]. Like these work, our method first transforms the source code to an intermediate representative, then performs static analysis. Unlike these work, our method does not rely on loops, because we focus on initialized but not used objects.

Trace and Statistical Analysis. Trace analysis is a way to detect performance bugs large scale systems. The Mystery Machine uses the information in different components of a request to construct causal relationships [4]. Aguilera et al. treat the components as "black-boxes" and obtain message traces [5]. Causal relationships can then be inferred and delay on specific node can be identified. While this method is powerful at a variety of situations, it is most effective in distributed systems. Also, it is not fully automated.

5. Discussion

In this section the limitations and future work of the four types of detection methods is discussed.

First, we detected performance bugs that relate to recreation. However, recreating a small portion of data structure may not cause performance issues. This type of performance bugs cannot be detected by our method. Programmer logic or source code inspection is needed to correct this type of performance bugs. For detecting the recreation of data structures, streams, and containers that cause performance issues, Jmap can be used to capture the number of existing instances of a class at runtime.

Second, we detected performance bugs that relate to unnecessary initialization. This method of static analysis is conservative. It's likely that some instances of a class are necessary, while others are not. It's also likely that different runs of the same program produce different "necessity" information. In order to detect unnecessary instances of a class, more space needs to be created when each instance is created. This space is used to record the method invocation events of each instance. When the instance is garbage collected, then JVM can decide whether this instance is necessary or not.

Third, our approaches to detect infinite loop are compatible with unit test. The tool we use, Cobertura, is itself a line coverage tool designed for unit test. However, these approaches are not suitable for production environment, because they have relatively large overhead. Also, the results could contain many false positives, therefore, human effort is needed to decide which ones are the actual bugs.

Fourth, in terms of detecting and mitigation unnecessary data in containers, the main limitation is that there is an assumption that the data of the fields in old classes and our newly defined classes is the same. If the field declarations have final as one of its specifier, then there is no such problem. However, if we can make sure that data does not change during the lifetime of the container then the range of optimizable classes can be extended further. Another limitation is the overhead of loading and creating a new class and copying data from the old class to the new one. If the new class does not have significantly less field attributes than the original class, then this optimization will not make significant difference.

6. Conclusion

Performance bugs are programming errors that degrade program performance. While existing methods can detect a variety of performance bugs, most of them are bound to loops or containers. Also, modern compilers perform sophisticated optimizations, but they cannot perceive enough runtime behavior.

In this paper, we analyzed 48 performance bugs from several popular open-source Java applications. We classified them into five categories and gave our detection suggestions based on their characteristics. We found that the most usual performance bug is related to heavy loop. We performed detection on four types of performance bugs, using both static analysis and dynamic analysis. We found that 1) Recreation of threads can cause less threads to be runnable. 2) Static analysis can filter out initialized but not used classes. 3) Static analysis can filter out unused attributes in container elements. 4) Statistical analysis of statement execution count can be used to infer infinite loop.

References

[1] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and detecting real-world performance bugs. ACM SIGPLAN Notices, 47 (6):77–88, 2012.

[2] Oswaldo Olivo, Isil Dillig, and Calvin Lin. 2015. Static detection of asymptotic performance bugs in collection traversals. In Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15). ACM, New York, NY, USA, 369-378.

[3] Adrian Nistor, Po-Chun Chang, Cosmin Radoi, and Shan Lu. 2015. Caramel: detecting and fixing performance problems that have non-intrusive fixes. In Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15), Vol. 1. IEEE Press, Piscataway, NJ, USA, 902-912.

[4] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F. Wenisch. 2014. The mystery machine: end-to-end performance analysis of large-scale internet services. In Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation (OSDI'14). USENIX Association, Berkeley, CA, USA, 217-231.

[5] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. 2003. Performance debugging for distributed systems of black boxes. SIGOPS Oper. Syst. Rev. 37, 5 (October 2003), 74-89.

[6] Ariel Eizenberg, Shiliang Hu, Gilles Pokam, and Joseph Devietti. 2016. Remix: online detection and repair of cache contention for the JVM. In Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16). ACM, New York, NY, USA, 251-265.

[7] Fastthread. Sudden CPU spike. Retrieved from http://fastthread.io