A GSoC 2015 Proposal

Detecting Redundant Operations with LLVM

Mingxing Zhang, *Tsinghua University*

An operation is classified as redundant if either *1)* it is unneeded thus omitting it will not change the output of the program; or *2)* it is repeated, which means the result of it is the same as a former run. The prevalent existences of these redundancies are a main source of performance bugs [3]. In this project, we intend to detect these bugs by *1)* identifying the redundant operations on instruction granularity via dynamic slicing; and then *2)* aggregating them into high-level blocks that worth fixing. In comparison with our method, the previous works [5, 6] on this topic only focus on detecting cacheable data (the repeated operations) and neglect the unneeded operations.

1. Introduction

A key reason why performance bugs escape so easily to production phase is the lack of reliable oracles, which means that it is usually hard for the developers to distinguish whether a performance bug is triggered or not. The profiler, which is the most commonly used method at present, are not effective enough because it only reports what takes time but not what wastes time, thus it may miss a performance bug if the buggy code is not slow compared to the rest part of the execution. However, such an oracle exists if we only focus on the bugs cause by redundant operations.

An operation is classified as redundant if either *1)* it is unneeded thus omitting it will not change the output of the program; or *2)* it is repeated, which means the result of it is the same as a former run. The time consumed by these redundant operations are simply wasted (although it does not mean that these wastage can be easily avoided). And more importantly, according to our investigation, the prevalent existences of them are a main source of performance bugs [3].

As an illustration, *Apache*#45464 gives an example of unneeded operation, in which the programmer passes an inappropriate parameter to $apr\_stat()$ thus unneeded data is calculated and causes a $10X$ slowdown in Apache servers. For the repeated operations, *Apache*#48778 shows a bug that can be avoided by memorizing the result of $NumberFormat.getInstance()$.

In this project, we intend to use LLVM to detect these redundant operations in three steps: *1)* identifying the redundant operations on instruction granularity via dynamic slicing; *2)* aggregating them into high-level blocks that worth fixing; and *3)* fixing the bugs by passing specific parameters or adding proper break/return/if statements to bypass the redundant code.

In comparison with our method, the previous works on this topic only focus on detecting cacheable data [5] or similar memory-access patterns [6], which are both a subset of the repeated operations. Thus they cannot detect the bugs caused by unneeded operations.

2. Tentative Method

Step 1: Identifying the redundant operations on instruction granularity.

Part 1: Identifying unneeded instructions.

1. Mark all the parameters of an output system call (e.g., print) as needed.
2. If a data is needed, the last efficacious[1] write instruction to it is needed.
3. If an instruction is needed,
   a) all the data it reads is needed;
   b) the control flow instruction that leads to its execution is needed.
4. An instruction is unneeded if it is not needed.

Part 2: Identifying repeated instructions.

An instruction is repeated if there exist K other instructions that:

1. are derived from the static instruction; and
2. read/write the same value.

Step 2: Aggregating them into high-level blocks that worth fixing.

According to our investigation, performance bugs caused by redundant operations can be classified into several different categories

1. The result of certain function is usually the same.
2. Only part of the function results are used in future.
3. The later part of a loop/function does not need to be executed.
4. Some operation does not need to be executed under certain condition.

Thus we may use specific rules to detect them respectively.

Step 3: Fixing the bugs manually.

***** All the information can be gathered by using static instrumenting implemented with LLVM.

3. Comparision with Related Works

Cachetor [5] is a dynamic analysis tool to pinpoint cacheable data values and operations producing these values. Thus it focus on detecting the bugs caused by ignoring cacheable objects (Type 5 in Appendix A). However, bugs cause by unneeded computations (Type 1-4 in Appendix A) are beyond its ability.

Toddler [6] is similar to Cachetor, which intends to detect repetitive and partially similar memory accesses across loop iterations. The intuition is that if a group of instructions repeatedly accesses similar memory values, they probably compute similar results. Compared to Cachetor, Toddler can detect not only cacheable function results but is still constrained in loops.

[1] A write is efficacious if the written value is different from the previous value

4. Preliminary Results

I've already implemented an inefficient prototype of the above method. As a proof-of-concept prototype, the current implementation use LLVM to instrument the main application and use Intel's PIN Tool [4] for instrumenting all the libraries and system calls, thus it will incur huge ovehead and cannot be used for complex applications. Moreover, it only uses very simple metrics for aggregating the instruction-level redundancies.

However, through investigating several simple applications with our prototype, we can somehow demonstrate the feasibility of the proposed algorithm. The preliminary evaluation results are given bellow.

4.1. Detecting Bugs

First, I've verified the prototype's bug-detecting ability by using some extracted bug-kernel (semantically equivalent to the original bug) of known bugs.

Second, I've also applied the prototype to several small applications and found some new bugs. For example, 6 bug sites (two useless memset, two cacheable function result, and two unused initializing) are found in Aget [1] and the patch is merged by the corresponding developers.

4.2. Overhead

As for the overhead, in our implementation we only need to instrumenting *1)* every memory access for computing memory dependencies; *2)* the ending of each basic block for computing control dependencies; and *3)* the starting and ending of every function for aligning traces. Thus it is promising that the instrumentation will only impose moderate overhead that is acceptable for in-house testing.

Due to the heay-weight dynamic instrumentation of PIN, the current prototype's overhead is $20\times - 30\times$ even for simple applications, and will generate gigabytes of trace files (most of them are infact unneeded). But, as listed in Section 5, we intend to replace PIN with LLVM by re-compiling the libraries and write specific instrumenting functions for every output system call respectively. This optimization will surely improve the performance a lot. Although I cannot give the exact overhead of the tool after the optimization, it should be similar to a former implementation of our AI tool [7], since they instrument the same set of instructions. Specifically, it should be about $2\times$ for desktop applications such as PBzip2; $5\times - 7\times$ for server applications Apache and MySQL; and $20 + \times$ for computation-intensive applications (e.g., FFT in the SPLASH-2 benchmark suites), thus is acceptable since the tool is only used in the in-house testing phase.

5. Plan

There are several things that I plain to do for implementing this redundancy-detecting tool based on the current prototype.

1. Since the current implementation does the instrumentation via opt, we should first reimplemented it with libLTO. This will facilitate the process of applying our tool on large software. (1 Week)

2. Currently PIN is used for obtaining the memory accessed while system calls. Although it is easy to use, dynamic instrumentation is notorious for its big overhead. Thus we should replace this part with instrumenting every output system call respectively. (2 Weeks)

3. The grouping step is now implemented by calculating the average ratio of redundant instructions of each call-site (call stack + calling instruction) and loop. Other metrics should also be attempted. (3 Weeks)

4. Verifying the usefulness of our tool with known bugs [3]. (2 Week)

5. Apply the tool on open-sourced software (e.g., Apache, MySQL) for detecting new bugs. (3 Weeks).

6. Scrub code, and write documents. (1 Week)

6. About Me

In this section, I'd like to introduce myself briefly. I'm a 3rd year PhD student from Tsinghua University, China. And my research area is mainly focus on software reliability and distributed computing, which means that I've used LLVM in many of my past and on-gong projects for detecting and tolerating bugs.

As an illustration, we have proposed a novel technique named Anticipating Invariant (AI) [7], which can anticipate concurrency bugs before any irreversible changes have been made. Based on it, we implemented a LLVM-based tool (about 2k loc) to tolerate concurrency bugs on-the-fly. Experiments with 35 real-world concurrency bugs demonstrate that AI is capable of detecting and tolerating most types of concurrency bugs, including both atomicity and order violations. We also evaluate AI with 6 representative parallel programs. Results show that AI incurs negligible overhead ($< 1\%$) for many nontrivial desktop and server applications. And its slowdown on computation-intensive programs can be reduced to about $2X$ after using the bias instrumentation. The paper is published in FSE 2014 and won a SIGSOFT distinguished paper award [2]. The source code of our implementation is also available at http://james0zan.github.io/AI.html. More importantly, the implementation of AI uses the same technique (static instrumentation) that will be used in this project, thus I do have some experiences on writing this kind of code.

I have also done a 3-months internship at Google NYC on improving database testing and a 9-months internship at Microsoft Research Asia (MSRA) for implementing a distributed (MPI-based) L-BFGS library. And I have got an excellent assessment for my MSRA internship. Moreover, I've visited the Columbia University for 6 months and working on a performance bug detection project, under the supervising of Prof. Junfeng Yang.

My email is james0zan@gmail.com and my homepage is http://james0zan.github.io/.

A. Categories of Redundant Operations

According to our investigation, the performance bugs caused by redundant operations can mainly be partitioned into 5 categories. In the following of this appendix, I will give an example of each category one by one.

A.1. Type 1: Only part of the function result are used in future

In Apache, *apr_stat*() is used to get information for a given file. The first parameter of this function is a apr_finfo_t struct, which is used to hold the return value. And the third parameter (a bit flag) is used to tell *apr_stat*() which information is required by the caller. If the caller uses some specific value as the third parameter (e.g., APR_FINFO_DEV), only some fields of the apr_finfo_t struct will be filled.

However, this feature is not fully explored by the developers. A total of 5 different bugs are reported in *Apache*#45464 and *Apache*#50878. They use APR_FINFO_NORM (i.e., all the fields should be filled) as the third parameter, which causes *apr_stat*() to return redundant values that are not used in the calling context, and brings obvious delay.

The corresponding patch is given bellow:

```
+++ modules/dav/fs/repos.c
-       rv = apr_stat(&dst_state_finfo, dst,
-                     APR_FINFO_NORM, p);
+       rv = apr_stat(&dst_state_finfo, dst,
+                     APR_FINFO_TYPE | APR_FINFO_DEV,
+                     p);
```

A.2. Type 2: The later part of a loop do not to be executed

In method *maybe_deduce_size_from_array_init*() of gcc the loop should break immediately after $failure$ is set to 1, since all the later iterations do not perform any useful work. This kind of missing break is prevalent, a total of 29 bugs are reported in GCC and Mozilla (e.g., GCC#57812, $Mozilla$#897258).

The corresponding patch is given bellow:

```
--- gcc/cp/decl.c    (revision 200588)
+++ gcc/cp/decl.c    (working copy)
  FOR_EACH_VEC_SAFE_ELT (v, i, ce)
    if (!check_array_designated_initializer (ce, i))
-     failure = 1;
+   {
+     failure = 1;
+     break;
+   }
```

A.3. Type 3: Some operation does not need to be executed under certain condition

In *Mozilla*#855464 the call of *WidthToClearPastFloats* in *ClearFloats*() is somewhat expensive, but it can be avoided if *floatAvailableSpace* is empty.

The corresponding patch is given bellow:

```
--- a/layout/generic/nsBlockReflowState.cpp
+++ b/layout/generic/nsBlockReflowState.cpp
  for (;;) {
    ...
+   if (!floatAvailableSpace.mHasFloats) break;
    ...
  }
```

A.4. Type 4: Inefficacious write

As an illustration, *Mozilla*#741027 reports a performance bug caused by computing the variable $m_t mpDownloadFile$ twice.

The corresponding patch is given bellow:

```
--- a/mailnews/local/src/nsPop3Sink.cpp
+++ b/mailnews/local/src/nsPop3Sink.cpp
-     rv = tmpDownloadFile->CreateUnique(
-             nsIFile::NORMAL_FILE_TYPE, 00600);
-     NS_ENSURE_SUCCESS(rv, rv);
-     m_tmpDownloadFile =
-       do_QueryInterface(tmpDownloadFile, &rv);
+     if (!m_tmpDownloadFile) {
+       rv =
+           tmpDownloadFile->CreateUnique(
+             nsIFile::NORMAL_FILE_TYPE, 00600);
+       NS_ENSURE_SUCCESS(rv, rv);
+       m_tmpDownloadFile =
+         do_QueryInterface(tmpDownloadFile, &rv);
+     }
```

A.5. Type 5: Cacheable objects

This kind of bug is prevelant and hence notorious. For example, *Apache*#19101 reports a 15% speedup by pulling the call to *.size*() out of the loop.

References

[1] https://github.com/enderunix/aget/pull/2.

[2] http://fse22.gatech.edu/dpa.

[3] Apache#44408, TILES#521, WW#3577, Apache#48778, Apache#50792, PIVOT#708, WW#3581, HDFS#1676, ZOOKEEPER#1015, Apache#19101, Apache#50716, Mozilla#913310, Apache#45464, Apache#50878, Mozilla#855464, GCC#57786, Mozilla#897258, Mozilla#983570, etc.

[4] https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.

[5] Khanh Nguyen and Guoqing Xu. Cachetor: Detecting cacheable data to remove bloat. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013.

[6] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. Toddler: Detecting performance problems via similar memory-access patterns. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13.

[7] Mingxing Zhang, Yongwei Wu, Shan Lu, Shanxiang Qi, Jinglei Ren, and Weimin Zheng. Ai: A lightweight system for tolerating concurrency bugs. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014.