Efficient detection of dangling pointer error for C/C++ programs

Wenzhe Zhang
Science and Technology on Parallel and Distributed Laboratory, State Key Laboratory of High Performance Computing, State Key Laboratory of High-end Server & Storage Technology, College of Computer, National University of Defense Technology, Changsha, PR China
Email: zhangwenzhe@nudt.edu.cn

Abstract. Dangling pointer error is pervasive in C/C++ programs and it is very hard to detect. This paper introduces an efficient detector to detect dangling pointer error in C/C++ programs. By selectively leave some memory accesses unmonitored, our method could reduce the memory monitoring overhead and thus achieves better performance over previous methods. Experiments show that our method could achieve an average speed up of 9% over previous compiler instrumentation based method and more than 50% over previous page protection based method.

1. Introduction
C/C++ programming requires programmers to manually organize dynamic memory [1]. Dynamic objects (memory spaces) are allocated via dynamic memory allocation API such as malloc and then freed via free [2]. This program paradigm has been used in many decades and it is criticized as being error-prone [3]. Often, programs may access a memory location that has been already freed, leading to unpredictable program results [4]. Moreover, it is very hard to notice this error during programming or debugging because reasoning all the malloc and free is complex [4].

This error, accessing already freed memory location, we call it dangling pointer error in this paper. Figure 1 shows an example of it. The last two lines in the sample code access the memory location pointed by a, which has been already freed. It is hard to notice this error during programming or debugging because that accessing the freed memory location (as shown in the last two lines in Figure 1) may not trigger any fault. Programs can run normally but give the wrong output, or programs may fault at a place that is very far from the error point, which makes debugging very complex.

Previous work [4] that aims to detect dangling pointer error mainly rely on dynamic instrumentation [5-6] or compiler instrumentation [7] to monitor every memory access. Normally, dynamic instrumentation is not acceptable because of its high overhead (more than 10X of the program’s execution time). Compiler instrumentation method is more popular but it is also expensive for that it needs to insert a monitoring function call before every memory access. Recent work [4] proposed to give each object (allocated by malloc) a single page and rely on page protection to monitor all accesses to these objects. However, giving each object a single page will cause a huge waste of virtual memory space, leading to virtual memory explosion and much more TLB (Translation Lookaside Buffer) miss. For programs that make intensive memory allocation, its performance overhead is much worse than the method of compiler instrumentation [7].
Above all, current mature method for monitoring memory access and detecting dangling pointer detection remains to be compiler instrumentation. In this paper we introduce an optimized compiler instrumentation method to detect dangling pointer error and at the same time instrumenting less memory accesses to improve performance. Our idea is straightforward: we only monitor the memory accesses that are after the program’s first `free` operation. For all the memory accesses that are before the first `free` operation, there will be no dangling pointer error because at that time, no memory spaces are freed. To determine whether a memory access is before or after the first `free` operation, we developed a special pass based on LLVM [7-8], a popular compiler framework, to do analysis at the intermediate code level. Experiments show that by monitoring less memory accesses, we can achieve 9% performance gain compared with the normal compiler instrumentation method and more than 50% compared with the page protection based method.

There are also a lot of methods proposed to detect dangling pointer error and these methods are mainly at higher levels such as overriding pointer dereference operations or using smart pointers. Moreover, the fast development of managed runtime system such as Java virtual machine offers new way to catch or eliminate dangling pointers. However, all these methods could not be used for C programs, which we believe is widely used in today, is commodity software development. In addition, those methods may be proved efficient for the higher-level languages, but the higher-level languages (such as Java) themselves are not as efficient as C. Thus in this paper we aim to raise a solution for the dangling pointer problem that can be adopted widely. The previous work would not replace our work in this paper.

The rest of this paper is organized as follows: We give the design and implementation of our detection method in Section 2. We show the experiments in Section 3 and conclude in Section 4.

2. Efficient Detection for Dangling Pointer Error

In theory, detecting dangling pointer error calls for two processes: (1) collecting dynamic memory allocation (`malloc`) or de-allocation (`free`) information (mainly the memory region of each object) during execution. (2) monitoring each memory access at runtime and check if the memory access refers to an already freed memory location. In this section, we will discuss these two processes respectively.

2.1. Dynamic Memory Allocation & De-Allocation

In order to get all the `malloc` and `free` information of the upper applications, we provide a dynamic linked library that overrides the `malloc` and `free` interface [1]. Thus, all the `malloc` and `free` operations called by upper applications will first go into our library. The code of `malloc` and `free` are shown in Figure 2. In our `malloc` and `free` we mainly first record the corresponding information and then call the original `malloc` and `free` that are provided by the standard glibc [9]. We keep all this information during the lifetime of a process thus at any time we can know if a memory access refers to a valid space or freed space.
2.2. Memory Access Monitoring

We rely on the famous compiler framework LLVM [7-8] to achieve the goal of monitoring memory accesses in programs. LLVM first compiles the program’s source code into a special intermediate code, at which level we can do a lot of analysis and code transformation such as loop unwinding [10]. Then the intermediate code will be compiled into the final binary code for execution.

![Figure 2. Code of malloc and free.](image1)

```c
void * malloc(size_t n)
{
    void * ret = ori_malloc(n);
    //record the malloc information
    record_malloc_info(ret, n);
    return ret;
}

void free(void *p)
{
    size_t n = get_size(p);
    //record the free information
    record_free_info(p, n);
    ori_free(p);
}
```

![Figure 3. Example of compiler instrumentation.](image2)

Here we developed an LLVM pass to insert checking function call before the memory accesses we want to monitor. The idea is shown in Figure 3. First, the original program code is show in Figure 3(a). Previous compiler instrumentation method is shown in Figure 3(b). We can see it just blindly inserts a check function before every memory access to check if the memory access will trigger a dangling pointer error. Thus all the memory accesses (the 4 accesses to a, b, c, d) shown in the figure are monitored. Our method is shown in Figure 3(c). We argue that there are some memory accesses that do not need to be monitored because at that time, no memory has been freed and thus it is not possible to trigger a dangling pointer error. As shown in Figure 3(c), the memory access to a in the main function and the memory access to c in the foo function can be left unmonitored. Thus we can get performance gain by eliminating the check function before them.
In order to determine if a memory access is before or after the first free operation in the program, we do a simple static analysis at the intermediate code level. We start from the main function and try to find the first function call of the main function. If this function call is free, we just mark all the memory accesses before the free can be left unmonitored. On the other hand, if this function is not free, we then start scan the code in the function (the case shown in Figure 3). Then if we find a free in the function, we can do the mark accordingly (the foo function shown in Figure 3). We can see in this simple example our method reduce half of the memory accesses that need to be monitored and thus we are likely to get performance gain at run time.

2.3. Dangling Pointer Error Detection

Above all, at execution time we got all the information of which piece of memory is valid and which piece of memory has been freed. We are also able to monitor all relative memory accesses. Then detecting dangling pointer error is simple and straightforward. At each memory access, we check if this memory location has been freed or not. If so, we report this to the programmer.

3. Experiments

This paper introduces an efficient method to detect dangling pointer error in C/C++ programs. In this section we mainly test and show our method (we call OPT in this section) in comparison with previous state-of-the-art work: (1) compiler instrumentation based work (we call Baseline in this section) and (2) page protection based work [4] (we call PPB in this section). The benchmarks we chose are mainly allocation intensive benchmarks from the olden benchmark suite [11]. They all make intense memory allocation and de-allocation, which is perfect to test and show our system.

![Figure 4. Performance Comparison.](image)

Figure 4 shows the result. First, as we can see, the previous page protection method (PPB) achieves very poor performance. This is because it gives each object a single page. When the benchmarks make intense memory allocation, it will use much more virtual space than it will needs, causing severe memory bloating. This extensive use of virtual memory space will cause sever TLB miss, leading to poor performance.

Second, we can see compared with previous compiler instrumentation method, our method achieves better performance on all benchmarks. This is mainly because that we selectively leave some memory accesses unmonitored. Moreover, the later a program calls free, the more benefit we can get. The case is shown in the benchmark perimeter and mst.

Finally, our method averagely achieves 9% performance gain over baseline and more than 50% performance gain over PPB.
4. Conclusion
This paper introduces an efficient detector to detect dangling pointer error in C/C++ programs. By selectively leave some memory accesses unmonitored, our method could reduce the memory monitoring overhead and thus achieves better performance over previous methods. Experiments show that our method could achieve an average speed up of 9% over previous compiler instrumentation based method and 50% over previous page protection based method.

5. Acknowledgments
The work is partially supported by the The National Key Research and Development Program of China (No. 2016YFB0200401), by program for New Century Excellent Talents in University, by National Science Foundation (NSF) China 61402492, 61402486, 61379146, by the laboratory pre-research fund (9140C810106150C81001).

6. References
[1] Berger, E. D., McKinley, K. S., Blumofe, R. D., & Wilson, P. R. (2000). Hoard: A scalable memory allocator for multithreaded applications. ACM Sigplan Notices, 35(11), 117-128.
[2] Kanev, S., Xi, S. L., Wei, G. Y., & Brooks, D. (2017, April). Mallacc: Accelerating Memory Allocation. In Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (pp. 33-45). ACM.
[3] Gay, D., Ennals, R., & Brewer, E. (2007, October). Safe manual memory management. In Proceedings of the 6th international symposium on Memory management (pp. 2-14). ACM.
[4] Dhurjati, D., & Adve, V. (2006, June). Efficiently detecting all dangling pointer uses in production servers. In Dependable Systems and Networks, 2006. DSN 2006. International Conference on (pp. 269-280). IEEE.
[5] Seward, J., Nethercote, N., & Fitzhardinge, J. (2004). Valgrind, an open-source memory debugger for x86-GNU/Linux. URL: http://www.ukuug.org/events/linux2002/papers/html/valgrind.
[6] Luk, C. K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G. ... & Hazelwood, K. (2005, June). Pin: building customized program analysis tools with dynamic instrumentation. In Acmsigplan notices (Vol. 40, No. 6, pp. 190-200). ACM.
[7] Lattner, C., & Adve, V. (2004, March). LLVM: A compilation framework for lifelong program analysis & transformation. In Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization (p. 75). IEEE Computer Society.
[8] Lattner, C. (2008, May). LLVM and Clang: Next generation compiler technology. In The BSD Conference (pp. 1-2).
[9] Kegel, D. (2007). Building and testing gcc/glibc cross toolchains.
[10] Baskaran, M. M., Bondhugula, U., Krishnamoorthy, S., Ramanujam, J., Rountev, A., & Sadayappan, P. (2008, June). A compiler framework for optimization of affine loop nests for GPGPUs. In Proceedings of the 22nd annual international conference on Supercomputing (pp. 225-234). ACM.
[11] CARLISLE, M. Olden benchmark suite. 1995-01-03[2007-06-30]. http://www.cs.princeton.edu/~mcc/olden.html.